

IBM Parallel Environment for AIX 5L



# MPI Programming Guide

*Version 4 Release 2*



IBM Parallel Environment for AIX 5L



# MPI Programming Guide

*Version 4 Release 2*

**Note**

Before using this information and the product it supports, read the information in "Notices" on page 223.

**Third Edition (April 2005)**

This edition applies to Version 4, Release 2 of IBM Parallel Environment for AIX 5L (product number 5765-F83) and to all subsequent releases and modifications until otherwise indicated in new editions. This edition replaces SA22-7945-01. Significant changes or additions to the text and illustrations are indicated by a vertical line ( | ) to the left of the change.

Order publications through your IBM representative or the IBM branch office serving your locality. Publications are not stocked at the address given below.

IBM welcomes your comments. A form for your comments appears at the back of this publication. If the form has been removed, address your comments to:

IBM Corporation, Department 55JA, Mail Station P384  
2455 South Road  
Poughkeepsie, NY 12601-5400  
United States of America

FAX (United States and Canada): 1+845+432-9405

FAX (Other Countries) Your International Access Code +1+845+432-9405

IBMLink (United States customers only): IBMUSM10(MHVRCFS)

Internet: mhvrfs@us.ibm.com

If you would like a reply, be sure to include your name, address, telephone number, or FAX number.

Make sure to include the following in your comment or note:

- Title and order number of this book
- Page number or topic related to your comment

When you send information to IBM, you grant IBM a nonexclusive right to use or distribute the information in any way it believes appropriate without incurring any obligation to you.

© Copyright International Business Machines Corporation 1993, 2005. All rights reserved.

US Government Users Restricted Rights – Use, duplication or disclosure restricted by GSA ADP Schedule Contract with IBM Corp.

---

# Contents

<b>Tables . . . . .</b>	<b>vii</b>
-------------------------	------------

<b>About this book . . . . .</b>	<b>ix</b>
----------------------------------	-----------

Who should read this book . . . . .	ix
How this book is organized . . . . .	ix
Conventions and terminology used in this book . . . . .	x
Abbreviated names . . . . .	x
Prerequisite and related information . . . . .	xi
Using LookAt to look up message explanations . . . . .	xi
How to send your comments . . . . .	xii
National language support (NLS) . . . . .	xii
Summary of changes for Parallel Environment 4.2 . . . . .	xii

<b>  Chapter 1. Performance Considerations for the MPI Library . . . . .</b>	<b>1</b>
--	----------

Message transport mechanisms . . . . .	1
Shared memory considerations . . . . .	2
MPI IP performance . . . . .	2
User Space considerations . . . . .	3
MPI point-to-point communications . . . . .	3
Eager messages . . . . .	3
Rendezvous messages . . . . .	5
Polling and single thread considerations . . . . .	6
LAPI send side copy. . . . .	7
Striping . . . . .	8
Remote Direct Memory Access (RDMA) considerations . . . . .	9
Other considerations. . . . .	9

<b>Chapter 2. Profiling message passing . . . . .</b>	<b>11</b>
---	-----------

AIX profiling . . . . .	11
MPI nameshift profiling . . . . .	11
MPI Nameshift profiling procedure . . . . .	11

<b>Chapter 3. Using shared memory . . . . .</b>	<b>15</b>
---	-----------

Point-to-point communications . . . . .	15
Collective communications . . . . .	15
Shared memory performance considerations . . . . .	16
Reclaiming shared memory . . . . .	16
Using POE with multiple Ethernet adapters and shared memory . . . . .	16

<b>Chapter 4. Performing parallel I/O with MPI . . . . .</b>	<b>19</b>
--	-----------

Definition of MPI-IO . . . . .	19
Features of MPI-IO . . . . .	19
Considerations for MPI-IO . . . . .	20
MPI-IO API user tasks. . . . .	20
Working with files . . . . .	21
Error handling . . . . .	22
Working with Info objects . . . . .	23
Using datatype constructors . . . . .	24
Setting the size of the data buffer . . . . .	24
MPI-IO file inter-operability . . . . .	24

<b>Chapter 5. Programming considerations for user applications in POE . . . . .</b>	<b>25</b>
---	-----------

The MPI library . . . . .	25
The signal library has been removed . . . . .	25

Parallel Operating Environment overview . . . . .	25
POE user limits . . . . .	26
Exit status . . . . .	26
POE job step function . . . . .	27
POE additions to the user executable . . . . .	27
Signal handlers . . . . .	28
Handling AIX signals . . . . .	28
Do not hard code file descriptor numbers . . . . .	29
Termination of a parallel job . . . . .	29
Do not run your program as root . . . . .	30
AIX function limitations . . . . .	30
Shell execution . . . . .	30
Do not rewind STDIN, STDOUT, or STDERR . . . . .	30
Do not match blocking and non-blocking collectives. . . . .	30
Passing string arguments to your program correctly . . . . .	31
POE argument limits . . . . .	31
Network tuning considerations . . . . .	31
Standard I/O requires special attention . . . . .	32
Reserved environment variables . . . . .	33
AIX message catalog considerations . . . . .	33
Language bindings . . . . .	33
Available virtual memory segments . . . . .	34
Using a switch clock as a time source. . . . .	34
Running applications with large numbers of tasks . . . . .	35
Running POE with MALLOCDEBUG. . . . .	35
Threaded programming . . . . .	36
Running single threaded applications. . . . .	36
POE gets control first and handles task initialization. . . . .	37
Limitations in setting the thread stack size . . . . .	37
Forks are limited . . . . .	37
Thread-safe libraries . . . . .	37
Program and thread termination . . . . .	37
Order requirement for system includes . . . . .	38
Using MPI_INIT or MPI_INIT_THREAD . . . . .	38
Collective communication calls . . . . .	38
Support for M:N threads . . . . .	38
Checkpoint and restart limitations. . . . .	39
64-bit application considerations . . . . .	42
MPI_WAIT_MODE: the nopoll option . . . . .	43
Mixed parallelism with MPI and threads . . . . .	43
Using MPI and LAPI in the same program . . . . .	43
Differences between MPI in PE 3.2 and PE Version 4 . . . . .	44
Differences between MPI in PE 4.1 and PE 4.2. . . . .	45
Other differences . . . . .	45
POE-supplied threads . . . . .	45
<b>Chapter 6. Using error handlers . . . . .</b>	<b>47</b>
Predefined error handler for C++ . . . . .	47
<b>Chapter 7. Predefined MPI datatypes . . . . .</b>	<b>49</b>
Special purpose datatypes . . . . .	49
Datatypes for C language bindings . . . . .	49
Datatypes for FORTRAN language bindings . . . . .	50
Datatypes for reduction functions (C reduction types) . . . . .	50
Datatypes for reduction functions (FORTRAN reduction types) . . . . .	51
<b>Chapter 8. MPI reduction operations . . . . .</b>	<b>53</b>
Predefined operations . . . . .	53
Datatype arguments of reduction operations . . . . .	53
Valid datatypes for the op option . . . . .	54

Examples . . . . .	55
C example. . . . .	55
FORTRAN example . . . . .	55
<b>Chapter 9. C++ MPI constants. . . . .</b>	<b>57</b>
Error classes . . . . .	57
Maximum sizes . . . . .	58
Environment inquiry keys . . . . .	58
Predefined attribute keys . . . . .	59
Results of communicator and group comparisons. . . . .	59
Topologies. . . . .	59
File operation constants . . . . .	59
MPI-IO constants . . . . .	59
One-sided constants . . . . .	60
Combiner constants used for datatype decoding functions. . . . .	60
Assorted constants . . . . .	60
Collective constants. . . . .	60
Error handling specifiers . . . . .	60
Special datatypes for construction of derived datatypes. . . . .	61
Elementary datatypes (C and C++) . . . . .	61
Elementary datatypes (FORTRAN) . . . . .	61
Datatypes for reduction functions (C and C++) . . . . .	61
Datatypes for reduction functions (FORTRAN) . . . . .	61
Optional datatypes . . . . .	62
Collective operations . . . . .	62
Null handles . . . . .	62
Empty group . . . . .	62
Threads constants . . . . .	63
FORTRAN 90 datatype matching constants. . . . .	63
<b>Chapter 10. MPI size limits . . . . .</b>	<b>65</b>
System limits. . . . .	65
Maximum number of tasks and tasks per node . . . . .	67
<b>Chapter 11. POE environment variables and command-line flags. . . . .</b>	<b>69</b>
MP_BUFFER_MEM details . . . . .	82
<b>Chapter 12. Parallel utility subroutines . . . . .</b>	<b>87</b>
mpc_isatty. . . . .	89
MP_BANDWIDTH, mpc_bandwidth . . . . .	91
MP_DISABLEINTR, mpc_disableintr . . . . .	96
MP_ENABLEINTR, mpc_enableintr . . . . .	99
MP_FLUSH, mpc_flush . . . . .	102
MP_INIT_CKPT, mpc_init_ckpt . . . . .	104
MP_QUERYINTR, mpc_queryintr . . . . .	106
MP_QUERYINTRDELAY, mpc_queryintrdelay . . . . .	109
MP_SET_CKPT_CALLBACKS, mpc_set_ckpt_callbacks . . . . .	110
MP_SETINTRDELAY, mpc_setintrdelay. . . . .	113
MP_STATISTICS_WRITE, mpc_statistics_write . . . . .	114
MP_STATISTICS_ZERO, mpc_statistics_zero . . . . .	117
MP_STDOUT_MODE, mpc_stdout_mode . . . . .	118
MP_STDOUTMODE_QUERY, mpc_stdoutmode_query . . . . .	121
MP_UNSET_CKPT_CALLBACKS, mpc_unset_ckpt_callbacks . . . . .	123
pe_dbg_breakpoint . . . . .	125
pe_dbg_checkpnt . . . . .	131
pe_dbg_checkpnt_wait . . . . .	135
pe_dbg_getcrid. . . . .	137
pe_dbg_getrtid . . . . .	138
pe_dbg_getvtid. . . . .	139
pe_dbg_read_cr_errfile . . . . .	140

pe_dbg_restart . . . . .	141
<b>Chapter 13. Parallel task identification API subroutines . . . . .</b>	<b>145</b>
poe_master_tasks . . . . .	146
poe_task_info . . . . .	147
<b>Appendix A. MPE subroutine summary . . . . .</b>	<b>149</b>
<b>Appendix B. MPE subroutine bindings . . . . .</b>	<b>151</b>
Bindings for non-blocking collective communication . . . . .	151
<b>Appendix C. MPI subroutine and function summary. . . . .</b>	<b>155</b>
<b>Appendix D. MPI subroutine bindings . . . . .</b>	<b>175</b>
Bindings for collective communication . . . . .	175
Bindings for communicators . . . . .	179
Bindings for conversion functions . . . . .	182
Bindings for derived datatypes . . . . .	183
Bindings for environment management. . . . .	189
Bindings for external interfaces . . . . .	191
Bindings for group management . . . . .	193
Bindings for Info objects. . . . .	195
Bindings for memory allocation . . . . .	196
Bindings for MPI-IO . . . . .	197
Bindings for MPI_Status objects . . . . .	204
Bindings for one-sided communication. . . . .	205
Bindings for point-to-point communication . . . . .	208
Binding for profiling control . . . . .	213
Bindings for topologies . . . . .	214
<b>Appendix E. PE MPI buffer management for eager protocol . . . . .</b>	<b>217</b>
<b>Appendix F. Accessibility . . . . .</b>	<b>221</b>
Accessibility information . . . . .	221
Using assistive technologies . . . . .	221
<b>Notices . . . . .</b>	<b>223</b>
Trademarks . . . . .	225
Acknowledgments. . . . .	226
<b>Glossary . . . . .</b>	<b>227</b>
<b>Index . . . . .</b>	<b>235</b>

---

## Tables

	1. Conventions and terminology used in this book. . . . .	x
	2. Parallel Environment abbreviations . . . . .	x
	3. How the clock source is determined . . . . .	35
	4. POE/MPI/LAPI Thread Inventory. . . . .	46
	5. Special purpose datatypes. . . . .	49
	6. Datatypes for C language bindings. . . . .	49
	7. Datatypes for FORTRAN language bindings . . . . .	50
	8. Datatypes for reduction functions (C reduction types) . . . . .	50
	9. Datatypes for reduction functions (FORTRAN reduction types)) . . . . .	51
	10. Predefined reduction operations. . . . .	53
	11. Valid datatype arguments. . . . .	53
	12. Valid datatypes for the op option . . . . .	54
	13. MPI eager limits . . . . .	66
	14. Task limits for parallel jobs . . . . .	67
	15. POE environment variables and command-line flags for partition manager control. . . . .	70
	16. POE environment variables/command-line flags for job specification . . . . .	73
	17. POE environment variables/command-line flags for I/O control. . . . .	74
	18. POE environment variables/command-line flags for diagnostic information . . . . .	76
	19. POE environment variables and command-line flags for Message Passing Interface (MPI) . . . . .	76
	20. POE environment variables/command-line flags for corefile generation . . . . .	83
	21. Other POE environment variables/command-line flags . . . . .	83
	22. Parallel utility subroutines . . . . .	87
	23. MPE Subroutines . . . . .	149
	24. Bindings for non-blocking collective communication . . . . .	151
	25. MPI subroutines and functions. . . . .	155
	26. Bindings for collective communication . . . . .	175
	27. Bindings for communicators . . . . .	179
	28. Bindings for conversion functions. . . . .	182
	29. Bindings for derived datatypes. . . . .	183
	30. Bindings for environment management . . . . .	189
	31. Binding for external interfaces . . . . .	191
	32. Bindings for groups . . . . .	193
	33. Bindings for Info objects . . . . .	195
	34. Bindings for memory allocation . . . . .	196
	35. Bindings for MPI-IO . . . . .	197
	36. Bindings for MPI_Status objects . . . . .	205
	37. Bindings for one-sided communication . . . . .	205
	38. Bindings for point-to-point communication . . . . .	208
	39. Binding for profiling control . . . . .	214
	40. Bindings for topologies . . . . .	214



---

## About this book

This book provides information about parallel programming, as it relates to IBM®'s implementation of the Message Passing Interface (MPI) standard for Parallel Environment for AIX® (PE). References to RS/6000® SP™ or SP include currently supported IBM @server Cluster 1600 hardware.

All implemented function in the PE MPI product is designed to comply with the requirements of the Message Passing Interface Forum, *MPI: A Message-Passing Interface Standard, Version 1.1*, University of Tennessee, Knoxville, Tennessee, June 6, 1995 and *MPI-2: Extensions to the Message-Passing Interface*, University of Tennessee, Knoxville, Tennessee, July 18, 1997. The second volume includes a section identified as MPI 1.2, with clarifications and limited enhancements to MPI 1.1. It also contains the extensions identified as MPI 2.0. The three sections, MPI 1.1, MPI 1.2, and MPI 2.0 taken together constitute the current standard for MPI.

PE MPI provides support for all of MPI 1.1 and MPI 1.2. PE MPI also provides support for all of the MPI 2.0 enhancements, except the contents of the chapter titled "Process creation and management."

If you believe that PE MPI does not comply, in any way, with the MPI standard for the portions that are implemented, please contact IBM service.

---

## Who should read this book

This book is intended for experienced programmers who want to write parallel applications using the C, C++, or FORTRAN programming language. Readers of this book should know C, C++, and FORTRAN and should be familiar with AIX and UNIX® commands, file formats, and special files. They should also be familiar with the MPI concepts. In addition, readers should be familiar with distributed-memory machines.

---

## How this book is organized

This book is organized as follows:

- Chapter 1, "Performance Considerations for the MPI Library," on page 1.
- Chapter 2, "Profiling message passing," on page 11.
- Chapter 3, "Using shared memory," on page 15.
- Chapter 4, "Performing parallel I/O with MPI," on page 19.
- Chapter 5, "Programming considerations for user applications in POE," on page 25.
- Chapter 6, "Using error handlers," on page 47.
- Chapter 7, "Predefined MPI datatypes," on page 49.
- Chapter 8, "MPI reduction operations," on page 53.
- Chapter 9, "C++ MPI constants," on page 57.
- Chapter 10, "MPI size limits," on page 65.
- Chapter 11, "POE environment variables and command-line flags," on page 69.
- Chapter 12, "Parallel utility subroutines," on page 87.
- Chapter 13, "Parallel task identification API subroutines," on page 145.

- Appendix A, “MPE subroutine summary,” on page 149.
- Appendix B, “MPE subroutine bindings,” on page 151.
- Appendix C, “MPI subroutine and function summary,” on page 155.
- Appendix D, “MPI subroutine bindings,” on page 175.
- Appendix E, “PE MPI buffer management for eager protocol,” on page 217.

## Conventions and terminology used in this book

This book uses the following typographic conventions:

Table 1. Conventions and terminology used in this book

Convention	Usage
<b>bold</b>	<b>Bold</b> words or characters represent system elements that you must use literally, such as: command names, file names, flag names, path names, PE component names ( <b>pedb</b> , for example), and subroutines.
constant width	Examples and information that the system displays appear in constant-width typeface.
<i>italic</i>	<i>Italicized</i> words or characters represent variable values that you must supply.  <i>Italics</i> are also used for book titles, for the first use of a glossary term, and for general emphasis in text.
[item]	Used to indicate optional items.
<Key>	Used to indicate keys you press.

In addition to the highlighting conventions, this manual uses the following conventions when describing how to perform tasks.

User actions appear in uppercase boldface type. For example, if the action is to enter the **tool** command, this manual presents the instruction as:

**ENTER**  
**tool**

## Abbreviated names

Some of the abbreviated names used in this book follow.

Table 2. Parallel Environment abbreviations

Short Name	Full Name
AIX	Advanced Interactive Executive
CSM	Clusters Systems Management
CSS	communication subsystem
CTSEC	cluster-based security
DPCL	dynamic probe class library
dsh	distributed shell
GUI	graphical user interface
HDF	Hierarchical Data Format
IP	Internet Protocol
LAPI	Low-level Application Programming Interface
MPI	Message Passing Interface

Table 2. Parallel Environment abbreviations (continued)

Short Name	Full Name
PE	IBM Parallel Environment for AIX
PE MPI	IBM's implementation of the MPI standard for PE
PE MPI-IO	IBM's implementation of MPI I/O for PE
POE	parallel operating environment
pSeries®	IBM @server pSeries
PSSP	IBM Parallel System Support Programs for AIX
RISC	reduced instruction set computer
RSCT	Reliable Scalable Cluster Technology
rsh	remote shell
RS/6000	IBM RS/6000
SP	IBM RS/6000 SP
STDERR	standard error
STDIN	standard input
STDOUT	standard output

## Prerequisite and related information

The Parallel Environment library consists of:

- *IBM Parallel Environment for AIX: Introduction*, SA22-7947
- *IBM Parallel Environment for AIX: Installation*, GA22-7943
- *IBM Parallel Environment for AIX: Messages*, GA22-7944
- *IBM Parallel Environment for AIX: MPI Programming Guide*, SA22-7945
- *IBM Parallel Environment for AIX: MPI Subroutine Reference*, SA22-7946
- *IBM Parallel Environment for AIX: Operation and Use, Volume 1*, SA22-7948
- *IBM Parallel Environment for AIX: Operation and Use, Volume 2*, SA22-7949

To access the most recent Parallel Environment documentation in PDF and HTML format, refer to the IBM @server Cluster Information Center on the Web at:

<http://publib.boulder.ibm.com/infocenter/clresctr/index.jsp>

Both the current Parallel Environment books and earlier versions of the library are also available in PDF format from the IBM Publications Center Web site located at:

<http://www.ibm.com/shop/publications/order/>

It is easiest to locate a book in the IBM Publications Center by supplying the book's publication number. The publication number for each of the Parallel Environment books is listed after the book title in the preceding list.

## Using LookAt to look up message explanations

LookAt is an online facility that lets you look up explanations for most of the IBM messages you encounter, as well as for some system abends and codes. You can use LookAt from the following locations to find IBM message explanations for Clusters for AIX and Linux®:

- The Internet. You can access IBM message explanations directly from the LookAt Web site:  
<http://www.ibm.com/eserver/zseries/zos/bkserv/lookat/>
- Your wireless handheld device. You can use the LookAt Mobile Edition with a handheld device that has wireless access and an Internet browser (for example, Internet Explorer for Pocket PCs, Blazer, or Eudora for Palm OS, or Opera for Linux handheld devices). Link to the LookAt Mobile Edition from the LookAt Web site.

---

## How to send your comments

Your feedback is important in helping to provide the most accurate and high-quality information. If you have any comments about this book or any other PE documentation:

- Send your comments by e-mail to: [mhvrdfs@us.ibm.com](mailto:mhvrdfs@us.ibm.com)  
Be sure to include the name of the book, the part number of the book, the version of PE, and, if applicable, the specific location of the text you are commenting on (for example, a page number or table number).
- Fill out one of the forms at the back of this book and return it by mail, by fax, or by giving it to an IBM representative.

---

## National language support (NLS)

For national language support (NLS), all PE components and tools display messages that are located in externalized message catalogs. English versions of the message catalogs are shipped with the PE licensed program, but your site may be using its own translated message catalogs. The PE components use the AIX environment variable **NLSPATH** to find the appropriate message catalog. **NLSPATH** specifies a list of directories to search for message catalogs. The directories are searched, in the order listed, to locate the message catalog. In resolving the path to the message catalog, **NLSPATH** is affected by the values of the environment variables **LC\_MESSAGES** and **LANG**. If you get an error saying that a message catalog is not found and you want the default message catalog:

**ENTER**

```
export NLSPATH=/usr/lib/nls/msg/%L/%N
```

```
export LANG=C
```

The PE message catalogs are in English, and are located in the following directories:

```
/usr/lib/nls/msg/C
```

```
/usr/lib/nls/msg/En_US
```

```
/usr/lib/nls/msg/en_US
```

If your site is using its own translations of the message catalogs, consult your system administrator for the appropriate value of **NLSPATH** or **LANG**. For more information on NLS and message catalogs, see *AIX: General Programming Concepts: Writing and Debugging Programs*.

---

## Summary of changes for Parallel Environment 4.2

This release of IBM Parallel Environment for AIX contains a number of functional enhancements, including:

- Support for POWER3™, POWER4™, and POWER5™ servers running AIX 5L™ V5.2 or AIX 5L V5.3
- Support for IBM @server p5 servers and the High Performance Switch (HPS) with AIX 5L V5.2 only, and coexistence in a cluster managed by Cluster Systems Management (CSM)
- Remote Direct Memory Access (RDMA) for bulk data copy and transfer, and large contiguous messages, only on the HPS
- Support for striping of messages over multiple adapters attached to the pSeries HPS
- MPI support for 128 tasks per node using shared memory
- Support for LoadLeveler® performance improvements
- Support for up to 8192 tasks in a single job, with improved memory utilization for large jobs
- MPI collectives algorithm and optimization improvements
- MPI shared memory collectives use AIX 5L V5.3 cross-memory attachment enhancements
- Point-to-point messages in shared memory use AIX 5L V5.3 cross-memory attachment enhancements
- MPI/LAPI performance statistics
- The SP Switch is no longer supported
- PE 4.2 is the **last** release of PE that will support Parallel Systems Support Programs for AIX (PSSP), the SP Switch2, and POWER3 servers



---

## Chapter 1. Performance Considerations for the MPI Library

This chapter provides performance considerations for the MPI library, including the following topics:

- “Message transport mechanisms.”
- “MPI point-to-point communications” on page 3.
- “Polling and single thread considerations” on page 6.
- “LAPI send side copy” on page 7
- “Striping” on page 8.
- “Remote Direct Memory Access (RDMA) considerations” on page 9.
- “Other considerations” on page 9.

Performance of jobs using the MPI library can be affected by the setting of various environment variables. The complete list is provided in Chapter 11, “POE environment variables and command-line flags,” on page 69 and in *IBM Parallel Environment for AIX 5L: Operation and Use, Volume 1*. Programs that conform to the MPI standard should run correctly with any combination of environment variables within the supported ranges.

The defaults of these environment variables are generally set to optimize the performance of the User Space library for MPI programs with one task per processor, using blocking communication. Blocking communication includes sets of non-blocking send and receive calls followed immediately by wait or waitall, as well as explicitly blocking send and receive calls. Applications that use other programming styles, in particular those that do significant computation between posting non-blocking sends or receives and calling wait or waitall, may see a performance improvement if some of the environment variables are changed.

---

### Message transport mechanisms

The MPI Library conforms to the MPI-2 Standard, with the exception of the chapter on *Process Creation and Management*, which is not implemented.

The MPI library is a dynamically loaded shared object, whose symbols are linked into the user application. At run time, when MPI\_Init is called by the application program, the various environment variables are read and interpreted, and the underlying transport is initialized. Depending on the setting of the transport variable **MP\_EUILIB**, MPI initializes lower level protocol support for a User Space packet mode, or for a UDP/IP socket mode. By default, the shared memory mechanism for point-to-point messages (and in 64-bit applications, collective communication) is also initialized.

Three message transport mechanisms are supported:

#### **Shared memory**

Used for tasks on the same node (as processes under the same operating system image)

#### **UDP/IP**

Used for tasks on nodes connected with an IP network

**User Space** Used for tasks having windows allocated on various versions of IBM high speed interconnects such as the pSeries High Performance Switch

These topics are addressed in the following sections, in detail:

- “Shared memory considerations”
- “MPI IP performance.”
- “User Space considerations” on page 3.

## Shared memory considerations

An MPI job can use a combination of shared memory and UDP/IP message transport mechanisms, or a combination of shared memory and User Space message transport mechanisms, for intertask communication. An MPI job may not use a combination of UDP/IP and User Space message transport mechanisms.

Tasks on the same node can use operating system shared memory transport for point-to-point communication. Shared memory is used by default, but may be turned off with the environment variable **MP\_SHARED\_MEMORY**. In addition, 64-bit applications are provided an optimization where the MPI library uses shared memory directly for selected collective communications, rather than just mapping the collectives into point-to-point communications. The collective calls for which this optimization is provided include **MPI\_Barrier**, **MPI\_Reduce**, **MPI\_Bcast**, **MPI\_Allreduce** and others. This optimization is enabled by default, and disabled by setting environment variable **MP\_SHARED\_MEMORY** to **no**. For most programs, enabling the shared memory transport for point-to-point and collective calls provides better performance than using the network transport.

For more information on shared memory, see Chapter 3, “Using shared memory,” on page 15.

## MPI IP performance

MPI IP performance is affected by the socket-buffer sizes for sending and receiving UDP data. These are defined by two network tuning parameters **udp\_sendspace** and **udp\_recvspace**. When the buffer for sending data is too small and quickly becomes full, UDP data transfer can be delayed. When the buffer for receiving data is too small, incoming UDP data can be dropped due to insufficient buffer space, resulting in send-side retransmission and very poor performance.

LAPI, on which MPI is running, tries to increase the size of send and receive buffers to avoid this performance degradation. However, the buffer sizes, **udp\_sendspace** and **udp\_recvspace**, cannot be greater than another network tuning parameter **sb\_max**, which can be changed only with privileged access rights (usually root). For optimal performance, it is suggested that **sb\_max** be increased to a relatively large value. For example, increase **sb\_max** from the default of 1048576 to 8388608 before running MPI IP jobs.

The UDP/IP transport can be used on clustered servers where a high speed interconnect is not available, or can use the IP mode of the high speed interconnect, if desired. This transport is often useful for program development or initial testing, rather than production. Although this transport does not match User Space performance, it consumes only virtual adapter resources rather than limited real adapter resources.

MPI with UDP/IP transport should be viewed as an IP application for system performance tuning. This transport is selected by setting the environment variable **MP\_EUILIB** to **ip** (must be lower case). The user may set the UDP packet size using the environment variable **MP\_UDP\_PACKET\_SIZE**, which should be set slightly smaller than the MTU of the IP network being used. The **MP\_** environment variables described in the remainder of this chapter may also affect performance with the IP transport, but have generally been designed with the optimized User Space transport in mind.

Details on the network tuning parameters, such as their definitions and how to change their values, can be found in the man page for the AIX **no** command.

## User Space considerations

The User Space transport binds one or more real adapter resources (called User Space windows) to each MPI task. The number of windows available depends on adapter type, but it is common for systems fully loaded with production jobs to have every available window committed. User Space is selected by setting the environment variable **MP\_EUILIB** to **us** (must be lower case). This is the transport for which the MPI library is optimized.

The underlying transport for MPI is LAPI, which is packaged with AIX as part of the RSCT file set. LAPI provides a one-sided message passing API, with optimizations to support MPI. Except when dealing with applications that make both MPI and direct LAPI calls, or when considering compatibility of PE and RSCT levels, there is usually little need for the MPI user to be concerned about what is in the MPI layer and what is in the LAPI layer.

---

## MPI point-to-point communications

To understand the various environment variables, it is useful to describe briefly how MPI manages point-to-point messages. Parts of this management are now in the LAPI LLP (Lower Level Protocol), which provides a reliable message delivery layer and a mechanism for asynchronous progress in MPI. Because LAPI runs above an unreliable packet layer, LAPI must deal with detecting and retransmitting any lost packet.

An MPI application program sends a message using either a blocking or a non-blocking send. A send is considered locally complete when the blocking send call returns, or when the wait associated with the non-blocking send returns. MPI defines a standard send as one that may complete before the matching receive is posted, or can delay its completion until the matching receive is posted. This definition allows the MPI library to improve performance by managing small standard sends with **eager protocol** and larger ones with **rendezvous protocol**. A small message is one no larger than the **eager limit** setting.

The eager limit is set by the **MP\_EAGER\_LIMIT** environment variable or the **-eager\_limit** command-line flag. For more information on the **MP\_EAGER\_LIMIT** environment variable, see Chapter 11, "POE environment variables and command-line flags," on page 69, and Appendix E, "PE MPI buffer management for eager protocol," on page 217.

## Eager messages

An **eager send** passes its buffer pointer, communicator, destination, length, tag and datatype information to a LLP reliable message delivery function. If the message is small enough, it is copied from the user's buffer into a protocol managed buffer,

and the MPI send is marked complete. This makes the user's send buffer immediately available for reuse. A longer message is not copied, but is transmitted directly from the user's buffer. In this second case, the send cannot be marked complete until the data has reached the destination and the packets have been acknowledged. It is because either the message itself, or a copy of it, is preserved until it can be confirmed that all packets arrived safely, that the LLP can be considered reliable. The strategy of making temporary copies of small messages in case a retransmission is required preserves reliability while it reduces the time that a small MPI send must block.

Whenever a send is active, and at other convenient times such as during a blocking receive or wait, a message dispatcher is run. This dispatcher sends and receives messages, creating packets for and interpreting packets from the lower level packet driver (User Space or IP). Since UDP/IP and User Space are both unreliable packet transports (packets may be dropped during transport without an error being reported), the message dispatcher manages packet acknowledgment and retransmission with a **sliding window protocol**. This message dispatcher is also run on a hidden thread once every few hundred milliseconds and, if environment variable **MP\_CSS\_INTERRUPT** is set, upon notification of packet arrival.

On the receive side, there are two distinct cases:

- The eager message arrives before the matching receive is posted.
- The receive is posted before the eager message arrives.

When the message dispatcher recognizes the first packet of an inbound message, a header handler or **upcall** is invoked. This upcall is to a function within the MPI layer that searches a list of descriptors for posted but unmatched receives. If a match is found, the descriptor is unlinked from the unmatched receives list and data will be copied directly from the packets to the user buffer. The receive descriptor is marked by a second upcall (a completion handler), when the dispatcher detects the final packet so that the MPI application can recognize that the receive is complete.

If a receive is not found by the header handler upcall, an **early arrival buffer** is allocated by MPI and the message data will be copied to that buffer. A descriptor similar to a receive descriptor but containing a pointer to the early arrival buffer is added to an **early arrivals list**. When an application does make a receive call, the early arrivals list is searched. If a match is found:

1. The descriptor is unlinked from the early arrivals list.
2. Data is copied from the early arrival buffer to the user buffer.
3. The early arrival buffer is freed.
4. The descriptor (which is now associated with the receive) is marked so that the MPI application can recognize that the receive is complete.

The size of the early arrival buffer is controlled by the **MP\_BUFFER\_MEM** environment variable.

The difference between a blocking and non-blocking receive is that a blocking receive does not return until the descriptor is marked complete, whether the message is found as an early arrival or is sent later. A non-blocking receive leaves a descriptor in the posted receives list if no match is found, and returns. The subsequent wait blocks until the descriptor is marked complete.

The MPI standard requires that a send not complete until it is guaranteed that its data can be delivered to the receiver. For an eager send, this means the sender must know in advance that there is sufficient buffer space at the destination to cache the message if no posted receive is found. The PE MPI library accomplishes this by using a credit flow control mechanism. At initialization time, each source to destination pair is allocated a fixed, identical number of **message credits**. The number of credits per pair is calculated based on environment variables **MP\_EAGER\_LIMIT**, **MP\_BUFFER\_MEM**, and the total number of tasks in the job. An MPI task sends eagerly to a destination as long as it has credits for that destination, but it costs one credit to send a message. Each receiver has enough space in its early arrival buffer to cache the messages represented by all credits held by all possible senders.

If an eager message arrives and finds a match, the credit is freed immediately because the early arrival buffer space that it represents is not needed. If data must be buffered, the credit is tied up until the matching receive call is made, which allows the early arrival buffer to be freed. PE MPI returns message flow control credits by piggybacking them on some regular message going back to the sender, if possible. If credits pile up at the destination and there are no application messages going back, MPI must send a special purpose message to return the credits. For more information on the early arrival buffer and the environment variables, **MP\_EAGER\_LIMIT** and **MP\_BUFFER\_MEM**, see Chapter 11, “POE environment variables and command-line flags,” on page 69 and Appendix E, “PE MPI buffer management for eager protocol,” on page 217.

## Rendezvous messages

For a standard send, PE MPI makes the decision whether to use an **eager** or a **rendezvous** protocol based on the message length. For the standard `MPI_Send` and `MPI_Isend` calls, messages whose size is not greater than the eager limit are sent using eager protocol. Messages whose size is larger than the eager limit are sent using rendezvous protocol. Thus, small messages can be eagerly sent, and assuming that message credits are returned in a timely fashion, can continue to be sent using the mechanisms described above. For large messages, or small messages for which there are no message credits available, the message must be managed with a rendezvous protocol.

Recall the following:

- The MPI definition for standard send promises the user that the message data will be delivered whenever the matching receive is posted.
- Send side message completion is no indication that a matching receive was found.

The decision made by an MPI implementation of standard send, to use eager protocol in some cases and rendezvous protocol in other cases is based on a need to allocate and manage buffer space for preserving eagerly sent message data in the cases where there is no receive waiting. The MPI standard’s advice that a ‘safe’ programming style must not assume a standard send will return before a matching receive is found, is also based on the requirement that the MPI implementation preserve any message data that it sends eagerly.

Since a zero byte message has no message data to preserve, even an MPI implementation with no early arrival buffering should be able to complete a zero byte standard send at the send side, whether or not there is a matching receive. Thus, for PE MPI with **MP\_EAGER\_LIMIT** set to zero, a one byte standard send

will not complete until a matching receive is found, but a zero byte standard send will complete without waiting for a rendezvous to determine whether a receive is waiting.

A rendezvous message is sent in two stages:

1. A message envelope is sent containing the information needed for matching by the receiver, and a message ID that is unique to the sender. This envelope either matches a previously posted receive, or causes a descriptor to be put in the list of early arrivals just as for an eager early arrival. Because the message data has not been sent, no early arrival buffer is needed.

Whether the matching receive is found when the envelope arrives, or the receive call is made later and matches a descriptor in the early arrivals list, an 'OK to send' response goes back to the sender after the match. This 'OK to send' contains the ID by which the sender identifies the data to send, and also an ID unique to the destination that identifies the match that was found.

2. When the sender gets an 'OK to send' message, it sends the message data, along with the destination side ID that identifies the receive that had been matched. As the data arrives, it can be copied directly into the receive buffer that was already identified as the match.

Eager messages require only one trip across the transport, while rendezvous messages require three trips, but two of the trips are short, and the time is quickly amortized for large messages. Using the rendezvous protocol ensures that there is no need for temporary buffers to store the data, and no overhead from copying packets to temporary buffers and then on to user buffers.

---

## Polling and single thread considerations

A blocking send or receive, or an MPI wait call, causes MPI to invoke the message dispatcher in a polling loop, processing packets as available until the specified message is complete. This is generally the lowest latency programming model, since packets are processed on the calling thread as soon as they arrive. The MPI library also supports an interrupt mode, specified by the environment variable **MP\_CSS\_INTERRUPT**, which causes an interrupt whenever a message packet arrives at the receiving network port or window.

In User Space, this interrupt is implemented as an AIX dispatch of a service thread that is created within each task at initialization time and is waiting on such an event. This thread calls the message dispatcher to process the packet, including invoking any upcalls to MPI for message matching or completion. Thus, while packets are being processed, other user threads may continue to perform computations. This is particularly useful if there are otherwise idle processors on the node, but that situation is not common. It is more likely to be useful with algorithms that allow communication to be scheduled well before the data is needed, and have computations to be done using data that is already available from a prior set of communications.

If all the processors are busy, enabling interrupt mode causes thread context switching and contention for processors, which might cause the application to run slower than it would in polling mode.

The behavior of the MPI library during message polling can also be affected by the setting of the environment variable **MP\_WAIT\_MODE**. If set to **sleep** or **yield**, the blocked MPI thread sleeps or yields periodically to allow the AIX dispatcher to schedule other activity on the processor. This may be appropriate when the wait

call is part of a command processor thread. An alternate way of implementing this behavior is with an MPI test command and user-invoked sleep or yield (or some other mechanism to release a processor).

Environment variable **MP\_WAIT\_MODE** can also be set to **nopoll**, which polls the message dispatcher for a short time (less than one millisecond) and then goes into a thread wait for either an interrupt or a timer expiration. In general, if **MP\_WAIT\_MODE** is set to **nopoll**, it is suggested that **MP\_CSS\_INTERRUPT** be set to **yes**.

As mentioned above, packets are transferred during polling and when an interrupt is recognized (which invokes the message dispatcher). The message dispatcher is also invoked periodically, based on the AIX timer support. The time interval between brief polls of the message dispatcher is controlled by environment variable **MP\_POLLING\_INTERVAL**, specified in microseconds.

The MPI library supports multiple threads simultaneously issuing MPI calls, and provides appropriate internal locking to make sure that the library is thread safe with respect to these calls. If the application makes MPI calls on only one thread (or is a non-threaded program), and does not use the nonstandard **MPE\_I** nonblocking collectives, **MPI-IO**, or **MPI** one-sided features, the user may wish to skip the internal locking by setting the environment variable **MP\_SINGLE\_THREAD** to **yes**. Do not set **MP\_SINGLE\_THREAD** to **yes** unless you are **certain** that the application is single threaded.

---

## LAPI send side copy

Some applications may benefit from changing the parameters controlling the send side copy mechanism. Because the send side buffering occurs at the level below **MPI**, the effect as seen by an **MPI** user must allow for headers used by **MPI**. To help you understand this as an **MPI** user, we must discuss it from a **LAPI** perspective.

**LAPI** send side guarantees making a copy of any **LAPI** level message of up to 128 bytes, letting the send complete locally. An **MPI** message sent by an application will have a header (or envelope) pre-appended by **PE MPI** before being sent as a **LAPI** message. Therefore, the application message size from the **MPI** perspective is less than from the **LAPI** perspective. The message envelope is no larger than 32 bytes. **LAPI** also maintains a limited pool of retransmission buffers larger than 128 bytes. If the application message plus **MPI** envelope exceeds 128 bytes, but is small enough to fit a retransmission buffer, **LAPI** tries (but cannot guarantee) to copy it to a retransmission buffer, allowing the **MPI** send to complete locally.

The size of the retransmission buffers is controlled by the environment variable **MP\_REXMIT\_BUF\_SIZE**, defaulting to a **LAPI** level message size of 16352 bytes. The supported **MPI** application message size is reduced by the number of bytes needed for the **MPI** envelope, which is 24 bytes for a 32-bit executable, or 32 bytes for a 64-bit executable.

The number of retransmission buffers is controlled by the environment variable **MP\_REXMIT\_BUF\_CNT**. The retransmission buffers are pooled, and are not assigned to a particular destination, so the appropriate number of buffers to achieve a balance between performance gain and memory cost is affected by the nature of the application and the system load.

If the message is successfully copied to a retransmission buffer, the local completion of the MPI send is immediate. If the message is too large to fit in the retransmission buffer, or if all the retransmission buffers are full (awaiting packet acknowledgement from their destination), the send does not complete locally until all message data has been received by the destination and acknowledged. Programs that do a group of blocking sends of a large number of messages that are expected to be sent eagerly may benefit from increasing the number of retransmission buffers. If memory allocation is of special concern, applications should set the retransmission buffer size to be no larger than the MPI eager limit plus the size of the MPI header.

For more information on the **MP\_EAGER\_LIMIT** environment variable, see Chapter 11, “POE environment variables and command-line flags,” on page 69 and Appendix E, “PE MPI buffer management for eager protocol,” on page 217.

---

## Striping

With PE Version 4, protocol striping is supported for HPS switch adapters (striping, failover, and recovery are not supported over non-HPS adapters such as Gigabit Ethernet). If the windows (or UDP ports) are on multiple adapters and one adapter or link fails, the corresponding windows are closed and the remaining windows are used to send messages. When the adapter or link is restored (assuming that the node itself remains operational), the corresponding windows are added back to the list of windows used for striping.

Striping is enabled when multiple instances are selected for communication. On a multi-network system, one way to do this is by choosing the composite device (set environment variable **MP\_EUIDEVICE** to **sn\_all** or **csss**), which requests allocation of one window on each network available on the node. For a node with two adapter links in a configuration where each link is part of a separate network, the result is a window on each of the two networks. For short messages and messages using the User Space FIFO mechanism, the CPU and memory bandwidth limits for copying user buffer data to the User Space FIFO packet buffers for transmission limits the achievable communication performance. Therefore, striping user space FIFO messages provides no performance benefit other than possibly better load balancing of the message traffic between the two networks. However, striping messages that use the Remote Direct Memory Access (RDMA) or bulk transfer mechanism can result in significant performance gains, since the data transfer function is off-loaded to the adapters, and there is very little CPU involvement in the communication.

For single network configurations, striping, failover, and recovery can still be used by requesting multiple instances (setting the environment variable **MP\_INSTANCES** to a value greater than 1). However, unless the system is configured with multiple adapters on the network, and window resources are available on more than one adapter, failover and recovery is not necessarily possible, because both windows may end up on the same adapter. Similarly, improved striping performance using RDMA can be seen only if windows are allocated from multiple adapters on the single network.

There are some considerations that users of 32-bit applications must take into account before deciding to use the striping, failover, and recovery function. A 32-bit application is limited to 16 segments. The standard AIX memory model for 32-bit applications claims five of these, and expects the application to allocate up to eight segments (2 GB) for application data (the heap, specified with compile option **-bmaxdata**). For example, **-bmaxdata:0x80000000** allocates the maximum eight

segments, each of which is 256 MB. The communication subsystem takes an additional, variable number of segments, depending on options chosen at run time.

In some circumstances, for 32-bit applications the total demand for segments can be greater than 16 and a job will be unable to start, or will run with reduced performance. If your application is using a very large heap and you consider enabling striping, see section *User Space striping with failover* in the chapter *Managing POE jobs of IBM Parallel Environment for AIX 5L: Operation and Use, Volume 1*.

---

## Remote Direct Memory Access (RDMA) considerations

Some MPI applications benefit from the use of the bulk transfer mode. This transfer mode is enabled by setting the LoadLeveler keyword **@bulkxfer** to **yes** or setting the environment variable **MP\_USE\_BULK\_XFER** to **yes** for interactive jobs. This transparently causes portions of the user's virtual address space to be pinned and mapped to a communications adapter. The low level communication protocol will then use Remote Direct Memory Access (RDMA, also known as bulk transfer) to copy (pull) data from the send buffer to the receive buffer as part of the MPI receive. The minimum message size for which RDMA will be used can be adjusted by setting environment variable **MP\_BULK\_MIN\_MSG\_SIZE**.

This especially benefits applications that either transfer relatively large amounts of data (greater than 150 KB) in a single MPI call, or overlap computation and communication, since the CPU is no longer required to copy data. RDMA operations are considerably more efficient when large (16 MB) pages are used rather than small (4 KB) pages, especially for large transfers. In order to use the bulk transfer mode, the system administrator must enable RDMA communication and LoadLeveler must be configured to use RDMA. Not all communications adapters support RDMA.

For a quick overview of the RDMA feature, and the steps that a system administrator must take to enable or disable the RDMA feature, see *Switch Network Interface for @server pSeries High Performance Switch Guide and Reference*.

For information on using LoadLeveler with bulk data transfer, see these sections in *LoadLeveler: Using and Administering*:

- The chapter: *Configuring the LoadLeveler environment*, section *Enabling support for bulk data transfer*.
- The chapter: *Building and submitting jobs*, section *Using bulk data transfer*.

---

## Other considerations

The information provided earlier in this chapter, and the controlling variables, apply to most applications. There are a few others that are useful in special circumstances. These circumstances may be identified by setting the **MP\_STATISTICS** environment variable to **print** and examining the task statistics at the end of an MPI job.

### **MP\_ACK\_THRESH**

This environment variable changes the threshold for the update of the packet sliding window. Reducing the value causes more frequent update of the window, but generates additional message traffic.

### **MP\_CC\_SCRATCH\_BUFFER**

MPI collectives normally pick from more than one algorithm based on the

| impact of message size, task count, and other factors on expected performance.  
| Normally, the algorithm that is predicted to be fastest is selected, but in some  
| cases the preferred algorithm depends on PE MPI allocation of scratch buffer  
| space. This environment variable instructs PE to use the collective  
| communication algorithm that takes less or even no scratch buffer space, even  
| if this algorithm is predicted to be slower. Most applications have no reason to  
| use this variable.

#### | **MP\_RETRANSMIT\_INTERVAL**

| This environment variable changes the frequency of checking for  
| unacknowledged packets. Lowering this value too much generates more switch  
| traffic and can lead to an increase in dropped packets. The packet statistics are  
| part of the end of job statistics displayed when **MP\_STATISTICS** is set to  
| **print**.

#### | **MP\_PRIORITY**

| This environment variable causes the invocation of the PE co-scheduler  
| function, if it is enabled by the system administrator. The value of this  
| environment variable is highly application dependent.

#### | **MP\_TASK\_AFFINITY**

| This environment variable applies to nodes that have more than one multi-chip  
| module (MCM) under control by AIX. It forces tasks to run exclusively on one  
| MCM, which allows them to take advantage of the memory local to that MCM.  
| This applies to IBM POWER4 and IBM POWER5 servers. For more information,  
| see *Managing task affinity on large SMP nodes* in *IBM Parallel Environment for AIX*  
| *5L: Operation and Use, Volume 1*.

---

## Chapter 2. Profiling message passing

This chapter describes how to profile your program for message passing, including the following topics:

- “AIX profiling.”
- “MPI nameshift profiling.”

---

### AIX profiling

If you use the **gprof**, **prof**, or **xprofiler** command and the appropriate compiler command (such as **cc\_r** or **mpcc\_r**) with the **-p** or **-pg** flag, you can profile your program. For information about using:

- **cc\_r**, **gprof**, and **prof**, see *IBM Parallel Environment for AIX: Operation and Use, Volume 2*.
- **mpcc\_r** and related compiler commands, see *IBM Parallel Environment for AIX: Operation and Use, Volume 1*.
- **xprofiler**, which is part of the AIX operating system, see the *AIX: Performance Tools Guide and Reference*.

The message passing library is not enabled for **gprof** or **prof** profiling counts. You can obtain profiling information by using the nameshifted MPI functions provided.

---

### MPI nameshift profiling

To use nameshift profiling routines that are either written to the C bindings with an MPI program written in C, or that are written to the FORTRAN bindings with an MPI program written in FORTRAN, follow the steps in “MPI Nameshift profiling procedure.”

Programs that use the C MPI language bindings can easily create profiling libraries using the nameshifted interface.

- If you are both the creator and user of the profiling library and you are not using FORTRAN, follow steps 1 through 6. If you are using FORTRAN, follow steps 1 through 4, then steps 7 through 14.
- If you are the creator of the profiling library, follow steps 1 through 4. You also need to provide the user with the file created in step 2.
- If you are the user of the profiling library and you are not using FORTRAN, follow steps 5 and 6. If you are using FORTRAN, start at step 7. You will need to make sure that you have the file generated by the creator in step 2.

### MPI Nameshift profiling procedure

To perform MPI nameshift profiling, follow the appropriate steps:

1. Create a source file that contains profiling versions of all the MPI subroutines you want to profile. For example, create a source file called **myprof\_r.c** that contains the following code:

```
#include <pthread.h>
#include <stdio.h>
#include <mpi.h>
int MPI_Init(int *argc, char ***argv) {
    int rc;
```

```

printf("hello from profiling layer MPI_Init...\n");
rc = PMPI_Init(argc, argv);
printf("goodbye from profiling layer MPI_Init...\n");
return(rc);
}

```

2. Create an export file that contains all of the symbols your profiling library will export. Begin this file with the name of your profiling library and the name of the **.o** file that will contain the object code of your profiling routines. For example, create a file called **myprof\_r.exp** that contains this statement:

```
MPI_Init
```

3. Compile the source file that contains your profiling MPI routines. For example:

```
cc_r -c myprof_r.c -I/usr/lpp/ppe.poe/include
```

The **-I** flag defines the location of **mpi.h**.

4. Create a shared library called **libmyprof\_r.a** that contains the profiled versions, exporting their symbols and linking with the PE MPI library, using **myprof\_r.exp** as shown. For example:

```
ld -o newmyprof_r.o myprof_r.o -bM:SRE -H512 -T512 -bnoentry
  -bE:myprof_r.exp -lc -lmpi_r -L/usr/lpp/ppe.poe/lib -lthreads
ar rv libmyprof_r.a newmyprof_r.o
```

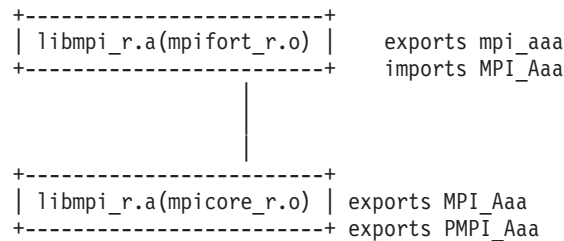
5. Link your user program:

```
mpcc_r -o test1 test1.c -L. -lmyprof_r
```

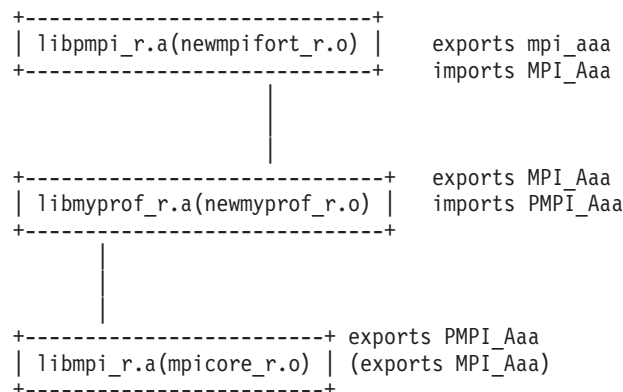
6. Run the resulting executable.

7. Programs that use the FORTRAN MPI language bindings need to do some additional steps to use the profiling libraries created above. This is because the FORTRAN bindings are contained in a separate shared object from the C bindings.

The shipped product has a library structure that looks like this:



You need to change it into the following structure by rebuilding the **mpifort\_r.o** shared object:



To do this, first extract **mpifort\_r.o** from **libmpi\_r.a**:

```
ar -xv /usr/lpp/ppe.poe/lib/libmpi_r.a mpifort_r.o
```

8. Then, construct a script to rebuild **mpifort\_r.o**, using the AIX **rtl\_enable** command:

```
rtl_enable -o newmpifort_r.o -s mpifort_r.o -L. -L/usr/lpp/ppe.poe/lib  
-lmyprof_r -lmpi_r -lc_r -lthreads
```

9. The **rtl\_enable** command creates a script called **mpifort\_r.sh** and import and export files that reflect the original binding with **libmpi\_r.a(mpicores\_r.o)**. To break this binding and rebind, remove the reference to the import file:

```
sed "s/-bI:mpifort_r.imp//" < mpifort_r.sh > mpifort_r.newsh
```

10. Make **mpifort\_r.newsh** executable and run it:

```
chmod +x mpifort_r.newsh  
mpifort_r.newsh
```

11. Archive the new shared object:

```
ar rv libmpi_r.a newmpifort_r.o
```

12. Create a program that uses an MPI function that you have profiled. For example, a file called **hwinit.f** could contain these statements:

```
c -----  
      program hwinit  
      include 'mpif.h'  
      integer fortterr  
  
c      call MPI_INIT(fortterr)  
  
c      Write comments to screen.  
c      write(6,*)'Hello from task '  
  
c      call MPI_FINALIZE(fortterr)  
  
c      stop  
      end  
  
c
```

13. Link your FORTRAN executable with the new library:

```
mpxlf_r -o hwinit hwinit.f -L. -lmpi_r
```

14. Run the resulting executable.



---

## Chapter 3. Using shared memory

This chapter addresses the use of shared memory and its performance considerations, including the following topics:

- “Point-to-point communications.”
- “Collective communications.”
- “Shared memory performance considerations” on page 16.
- “Reclaiming shared memory” on page 16.
- “Using POE with multiple Ethernet adapters and shared memory” on page 16.

---

### Point-to-point communications

MPI programs with more than one task on the same computing node may benefit from using shared memory to send messages between same node tasks.

This support is controlled by the **MP\_SHARED\_MEMORY** environment variable. The default setting is **yes**. In this case, shared memory is used for message passing. Message passing between tasks on different nodes continues to use User Space or IP protocol.

Setting this variable to **no** directs MPI to not use a shared-memory protocol for message passing between any two tasks of a job running on the same node.

For the 32-bit libraries, shared memory exploitation always allocates a 256 MB virtual memory address segment that is not available for any other use. Thus, programs that are already using all available segments cannot use this option. For more information, see “Available virtual memory segments” on page 34.

For 64-bit libraries, there are so many segments in the address space that there is no conflict between library and end user segment use.

Shared memory support is available for both IP and User Space MPI protocols. For programs on which *all* tasks are on the same node, shared memory is used exclusively for all MPI communication (unless **MP\_SHARED\_MEMORY** is set to **no**).

---

### Collective communications

With PE Version 4, the PE implementation of MPI also offers an optimization of certain collective communication routines. This optimization uses an additional shared memory segment. The collective communication optimization is available only to 64-bit executables, where segment registers are abundant. This optimization is controlled by the **MP\_SHARED\_MEMORY** environment variable.

For collectives in 64-bit executables that are enhanced to use shared memory, the algorithms used for smaller message sizes involve copying data from user buffers to scratch buffers in shared memory, and then allowing tasks that are interested in that data to work with the copy in shared memory. The algorithms used for larger messages involve exposing the user buffer itself to other tasks that have an interest in it. The effect is that for smaller messages, some tasks may return from a

collective call as soon as their data is copied to shared memory, sometimes before tasks needing access to the data even enter the collective operation.

For larger messages, the algorithms are more strongly synchronizing, because a task that directly exposes a user buffer to other tasks cannot return to the user until the interested tasks have completed their access to the data.

---

## Shared memory performance considerations

Be aware of these performance considerations:

1. The best performance is achieved when all message buffers are contiguous.
2. The large message support for some collectives involves exposing the memory of one task to the address space of another task. There is a limit of 4096 concurrent operations of this kind on a node. There is also a limit of 32 GB for the address range of a message that can use this technique.

If there are more than 4096 concurrent operations, or a buffer has an address range greater than 32 GB, performance abnormalities may be encountered.

This applies only to 64-bit executables, as discussed in the previous section, “Collective communications” on page 15.

3. A hang may occur if you match blocking and non-blocking collectives in the same application. For a full description, see “Do not match blocking and non-blocking collectives” on page 30.
4. 32-bit applications linked to use the maximum heap (8 segments) may not have enough available segments to effectively use shared memory for large messages. MPI will quietly use whatever resources are available, but performance may be impacted.

---

## Reclaiming shared memory

Occasionally, shared memory is not reclaimed. If this happens, you can use the **ipcrm** command, or contact the system administrator to reclaim the shared memory segments.

POE’s Partition Manager Daemon (PMD) attempts to clean up any allocated shared memory segments when a program exits normally. However, if a PMD process (named **pmdv4**) is killed with signals or with the **llcancel** command, shared memory segments may not be cleaned up properly. For this reason, when shared memory is used, users should not kill or cancel a PMD process.

---

## Using POE with multiple Ethernet adapters and shared memory

The following method can be used to run a non-LoadLeveler POE job that uses multiple Ethernet adapters and shared memory. If this method is not used for these jobs, POE cannot correctly determine which tasks are running on the same node, and shared memory key collisions will occur, resulting in unpredictable behavior. This method consists of an extra **poe** invocation before running the real POE job, and the use of a script that overrides an environment variable setting before executing the parallel task.

1. With **MP\_PROCS** set correctly in the environment (or with **-procs** set as part of the **poe** invocation), run  

```
poe hostname -stdoutmode ordered -ilevel 0 > hostnames
```

using the hostfile (either as host list in the directory where POE is run, or by specifying **MP\_HOSTFILE** or **-hostfile**) that contains the names of the Ethernet adapters.

2. If a shared file system is not used, copy the original hostfile and the **addr\_fix** script below to the nodes where the parallel tasks will run. The **addr\_fix** script must be copied to the directory with the same name as the current directory on the POE home node (from which you ran **poe** in step 1 on page 16.)
3. Run your real POE job with whatever settings you were using, except:
  - Use the hostnames file from step 1 on page 16 as the **MP\_HOSTFILE** or **-hostfile** that is specified to POE.
  - Set the environment variable **ADDR\_FIX\_HOSTNAME** to the name of the hostfile that contains the names of the Ethernet adapters, used in step 1 on page 16.
  - Instead of invoking the job as:  
`poe my_exec my_args poe_flags`

invoke it as:

```
poe ./addr_fix my_exec my_args poe_flags
```

The **addr\_fix** script follows.

```
=====
#!/bin/ksh93

# Determine file index based on taskid
my_index=`expr $MP_CHILD + 1`

# Index into the file to get the ethernet name that this task will run on.
my_name=`cat $ADDR_FIX_HOSTNAME | awk NR==$my_index'{print $0}'`

# Convert my_name to a dot decimal address.
my_addr=`host $my_name | awk '{print $3}' | tr ',' ' '`

# Set environment variable that MPI will use as address for IP communication
export MP_CHILD_INET_ADDR=@1:$my_addr,ip

# Execute what was passed in
$*
=====
```

This script assumes that striping is not used.

If LAPI is used, set **MP\_LAPI\_INET\_ADDR** in the script instead. If both MPI and LAPI are used, set both environment variables.



---

## Chapter 4. Performing parallel I/O with MPI

This chapter describes how to perform parallel I/O with MPI, including the following topics:

- “Definition of MPI-IO.”
- “Features of MPI-IO.”
- “Considerations for MPI-IO” on page 20.
- “MPI-IO API user tasks” on page 20.
- “MPI-IO file inter-operability” on page 24.

---

### Definition of MPI-IO

The I/O component of MPI-2, or *MPI-IO*, provides a set of interfaces that are aimed at performing portable and efficient parallel input and output operations.

MPI-IO allows a parallel program to express its I/O in a portable way that reflects the program’s inherent parallelism. MPI-IO uses many of the concepts already provided by MPI to express this parallelism. MPI datatypes are used to express the layout and partitioning of data, which is represented in a file shared by several tasks. An extension of the MPI communicator concept, referred to as an *MPI\_File*, is used to describe a set of tasks and a file that these tasks will use in some integrated manner. Collective operations on an *MPI\_File* allow efficient physical I/O on a data structure that is distributed across several tasks for computation, but possibly stored contiguously in the underlying file.

---

### Features of MPI-IO

The primary features of MPI-IO are:

1. **Portability:** As part of MPI-2, programs written to use MPI-IO must be portable across MPI-2 implementations and across hardware and software platforms. The PE MPI-IO implementation guarantees portability of object code on RS/6000 SP computers and clustered servers. The MPI-IO API ensures portability at the source code level.
2. **Versatility:** The PE MPI-IO implementation provides support for:
  - basic file manipulations (open, close, delete, sync)
  - get and set file attributes (view, size, group, mode, info)
  - blocking data access operations with explicit offsets (both independent and collective)
  - non-blocking data access operations with explicit offsets (independent only)
  - blocking and non-blocking data access operations with file pointers (individual and shared)
  - split collective data access operations
  - any derived datatype for memory and file mapping
  - file inter-operability through data representations (internal, external, user-defined)
  - atomic mode for data accesses.
3. **Robustness:** PE MPI-IO performs as robustly as possible in the event of error occurrences. Because the default behavior, as required by the MPI-2 standard, is

for I/O errors to return, PE MPI-IO tries to prevent any deadlock that might result from an I/O error returning. The intent of the "errors return" default is that the type of errors considered almost routine in doing I/O should not be fatal in MPI (for example, a "file not found" error).

However, deadlocks resulting from erroneous user codes cannot be entirely avoided. Users of MPI-IO routines should always check return codes and be prepared to terminate the job if the error is not one that the application can recover from.

An application that fails in trying to create a file, fails every time it tries to write, and fails again closing the file, will run to completion with no sign of a problem, if return codes are not checked. The common practice of ignoring return codes on MPI calls trusting MPI to trap the failure does not work with MPI-IO calls.

---

## Considerations for MPI-IO

MPI-IO will not operate if the **MP\_SINGLE\_THREAD** environment variable is set to **yes**. A call to **MPI\_INIT** with **MP\_SINGLE\_THREAD** set to **yes** is equivalent to what might be expected with a call to **MPI\_INIT\_THREAD** specifying **MPI\_THREAD\_FUNNELED**. A call with **MP\_SINGLE\_THREAD** set to **no** is equivalent to using **MPI\_THREAD\_MULTIPLE**. The default setting of **MP\_SINGLE\_THREAD** is **no**, therefore the default behavior of the threads library is **MPI\_THREAD\_MULTIPLE**.

**Note:** In PE MPI, thread behavior is determined before calling **MPI\_INIT** or **MPI\_INIT\_THREAD**. A call to **MPI\_INIT\_THREAD** with **MPI\_THREAD\_FUNNELED** will not actually mimic **MP\_SINGLE\_THREAD**.

MPI-IO is intended to be used with the IBM General Parallel File System (GPFS) for production use. File access through MPI-IO normally requires that a single GPFS file system image be available across all tasks of an MPI job. Shared file systems such as AFS® and NFS do not meet this requirement when used across multiple nodes. PE MPI-IO can be used for program development on any other file system that supports a POSIX interface (AFS, DFS™, JFS, or NFS) as long as all tasks run on a single node or workstation, but this is not expected to be a useful model for production use of MPI-IO.

In MPI-IO, whether an individual task performs I/O is **not** determined by whether that task issues MPI-IO calls. By default, MPI-IO performs I/O through an agent at each task of the job. I/O agents can be restricted to specific nodes by using an I/O node file. This should be done any time there is not a single GPFS file system available to all nodes on which tasks are to run. PE MPI-IO can be used without all tasks having access to a single file system image by using the **MP\_IONODEFILE** environment variable. See *IBM Parallel Environment for AIX: Operation and Use, Volume 1* for information about **MP\_IONODEFILE**.

---

## MPI-IO API user tasks

This section explains the following MPI-IO user tasks:

- "Working with files" on page 21.
- "Error handling" on page 22.
- "Working with Info objects" on page 23.
- "Using datatype constructors" on page 24.

- “Setting the size of the data buffer” on page 24.

## Working with files

This section explains MPI-IO file management tasks.

### Opening a file (MPI\_FILE\_OPEN)

When MPI-IO is used correctly, a file name will refer to the same file system at every task of the job, not just at every task that issues the MPI\_FILE\_OPEN. In one detectable error situation, a file will appear to be on different file system types. For example, a particular file could be visible to some tasks as a GPFS file and to others as NFS-mounted.

Use of a file that is local to (that is, distinct at) each task or node, is not valid and cannot be detected as an error by MPI-IO. Issuing MPI\_FILE\_OPEN on a file in `/tmp` may look valid to the MPI library, but will not produce valid results.

The default for `MP_CSS_INTERRUPT` is **no**. If you do not override the default, MPI-IO enables interrupts while files are open. If you have forced interrupts to **yes** or **no**, MPI-IO does not alter your selection.

MPI-IO depends on hidden threads that use MPI message passing. MPI-IO cannot be used with `MP_SINGLE_THREAD` set to **yes**.

For AFS, DFS, and NFS, MPI-IO uses file locking for all accesses by default. If other tasks on the same node share the file and also use file locking, file consistency is preserved. If the MPI\_FILE\_OPEN is done with mode `MPI_MODE_UNIQUE_OPEN`, file locking is not done.

For information about file hints, see MPI\_FILE\_OPEN in *IBM Parallel Environment for AIX: MPI Subroutine Reference*.

### Other file tasks

For information about the following file tasks, see *IBM Parallel Environment for AIX: MPI Subroutine Reference*.

- Closing a file (MPI\_FILE\_CLOSE)
- Deleting a file (MPI\_FILE\_DELETE)
- Resizing a file (MPI\_FILE\_SET\_SIZE)
- Preallocating space for a file (MPI\_FILE\_PREALLOCATE)
- Querying the size of a file (MPI\_FILE\_GET\_SIZE)
- Querying file parameters (MPI\_FILE\_GET\_AMODE, MPI\_FILE\_GET\_GROUP)
- Querying and setting file information (MPI\_FILE\_GET\_INFO, MPI\_FILE\_SET\_INFO)
- Querying and setting file views (MPI\_FILE\_GET\_VIEW, MPI\_FILE\_SET\_VIEW)
- Positioning (MPI\_FILE\_GET\_BYTE\_OFFSET, MPI\_FILE\_GET\_POSITION)
- Synchronizing (MPI\_FILE\_SYNC)
- Accessing data
  - Data access with explicit offsets:
    - MPI\_FILE\_READ\_AT
    - MPI\_FILE\_READ\_AT\_ALL
    - MPI\_FILE\_WRITE\_AT
    - MPI\_FILE\_WRITE\_AT\_ALL

- MPI\_FILE\_IREAD\_AT
- MPI\_FILE\_IWRITE\_AT
- Data access with individual file pointers:
  - MPI\_FILE\_READ
  - MPI\_FILE\_READ\_ALL
  - MPI\_FILE\_WRITE
  - MPI\_FILE\_WRITE\_ALL
  - MPI\_FILE\_IREAD
  - MPI\_FILE\_IWRITE
  - MPI\_FILE\_SEEK
- Data access with shared file pointers:
  - MPI\_FILE\_READ\_SHARED
  - MPI\_FILE\_WRITE\_SHARED
  - MPI\_FILE\_IREAD\_SHARED
  - MPI\_FILE\_IWRITE\_SHARED
  - MPI\_FILE\_READ\_ORDERED
  - MPI\_FILE\_WRITE\_ORDERED
  - MPI\_FILE\_SEEK
  - MPI\_FILE\_SEEK\_SHARED
- Split collective data access:
  - MPI\_FILE\_READ\_AT\_ALL\_BEGIN
  - MPI\_FILE\_READ\_AT\_ALL\_END
  - MPI\_FILE\_WRITE\_AT\_ALL\_BEGIN
  - MPI\_FILE\_WRITE\_AT\_ALL\_END
  - MPI\_FILE\_READ\_ALL\_BEGIN
  - MPI\_FILE\_READ\_ALL\_END
  - MPI\_FILE\_WRITE\_ALL\_BEGIN
  - MPI\_FILE\_WRITE\_ALL\_END
  - MPI\_FILE\_READ\_ORDERED\_BEGIN
  - MPI\_FILE\_READ\_ORDERED\_END
  - MPI\_FILE\_WRITE\_ORDERED\_BEGIN
  - MPI\_FILE\_WRITE\_ORDERED\_END

## Error handling

MPI-1 treated all errors as occurring in relation to some communicator. Many MPI-1 functions were passed a specific communicator, and for the rest it was assumed that the error context was MPI\_COMM\_WORLD. MPI-1 provided a default error handler named MPI\_ERRORS\_ARE\_FATAL for each communicator, and defined functions similar to those listed below for defining and attaching alternate error handlers.

The MPI-IO operations use an MPI\_File in much the way other MPI operations use an MPI\_Comm, except that the default error handler for MPI-IO operations is MPI\_ERRORS\_RETURN. The following functions are needed to allow error handlers to be defined and attached to MPI\_File objects:

- MPI\_FILE\_CREATE\_ERRHANDLER
- MPI\_FILE\_SET\_ERRHANDLER

- MPI\_FILE\_GET\_ERRHANDLER
- MPI\_FILE\_CALL\_ERRHANDLER

For information about these subroutines, see *IBM Parallel Environment for AIX: MPI Subroutine Reference*.

## Logging I/O errors

Set the **MP\_IO\_ERRLOG** environment variable to **yes** to indicate whether to turn on error logging for I/O operations. For example:

```
export MP_IO_ERRLOG=yes
```

turns on error logging. When an error occurs, a line of information will be logged in file `/tmp/mpi_io_errdump.app_name.userid.taskid`, recording the time the error occurs, the POSIX file system call involved, the file descriptor, and the returned error number.

## Working with Info objects

The MPI-2 standard provides the following Info functions as a means for a user to construct a set of hints and pass these hints to some MPI-IO operations:

- MPI\_INFO\_CREATE
- MPI\_INFO\_DELETE
- MPI\_INFO\_DUP
- MPI\_INFO\_FREE
- MPI\_INFO\_GET
- MPI\_INFO\_GET\_NKEYS
- MPI\_INFO\_GET\_NTHKEY
- MPI\_INFO\_SET
- MPI\_INFO\_GET\_VALUELEN

An *Info object* is an opaque object consisting of zero or more (key,value) pairs. Info objects are the means by which users provide hints to the implementation about things like the structure of the application or the type of expected file accesses. In MPI-2, the APIs that use Info objects span MPI-IO, MPI one-sided, and dynamic tasks. Both key and value are specified as strings, but the value may actually represent an integer, boolean or other datatype. Some keys are reserved by MPI, and others may be defined by the implementation. The implementation defined keys should use a distinct prefix which other implementations would be expected to avoid. All PE MPI hints begin with **IBM\_** (see *MPI\_FILE\_OPEN* in *IBM Parallel Environment for AIX: MPI Subroutine Reference*). The MPI-2 requirement that hints, valid or not, cannot change the semantics of a program limits the risks from misunderstood hints.

By default, Info objects in PE MPI accept only PE MPI recognized keys. This allows a program to identify whether a given key is understood. If the key is not understood, an attempt to place it in an Info object will be ignored. An attempt to retrieve the key will find no key/value present. The environment variable **MP\_HINTS\_FILTERED** set to **no** will cause Info operations to accept arbitrary (key, value) pairs. You will need to turn off hint filtering if your application, or some non-MPI library it is using, depends on MPI Info objects to cache and retrieve its own (key, value) pairs.

## Using datatype constructors

The following type constructors are provided as a means for MPI programs to describe the data layout in a file and relate that layout to memory data which is distributed across a set of tasks. The functions exist only for MPI-IO.

- `MPI_TYPE_CREATE_DARRAY`
- `MPI_TYPE_CREATE_SUBARRAY`

## Setting the size of the data buffer

Set the `MP_IO_BUFFER_SIZE` environment variable to indicate the default size of the data buffers used by the MPI-IO agents. For example:

```
export MP_IO_BUFFER_SIZE=16M
```

sets the default size of the MPI-IO data buffer to 16 MB. The default value of this environment variable is the number of bytes corresponding to 16 file blocks. This value depends on the block size associated with the file system storing the file.

Valid values are any positive size up to 128 MB. The size can be expressed as a number of bytes, as a number of kilobytes (1024 bytes), using **k** or **K** as a suffix, or as a number of megabytes (1024\*1024 bytes), using **m** or **M** as a suffix. If necessary, PE MPI rounds the size up, to correspond to an integral number of file system blocks.

---

## MPI-IO file inter-operability

For information about the following file inter-operability topics, see *IBM Parallel Environment for AIX: MPI Subroutine Reference* and the *MPI-2 Standard*:

- Datatypes (`MPI_FILE_GET_TYPE_EXTENT`)
- External data representation (`external32`)
- User-defined data representations (`MPI_REGISTER_DATAREP`)
  - Extent callback
  - Datarep conversion functions
- Matching data representations

For information about the following topics, see the *MPI-2 Standard*:

- Consistency and semantics
  - File consistency
  - Random access versus sequential files
  - Progress
  - Collective file operations
  - Type matching
  - Miscellaneous clarifications
  - `MPI_Offset` Type
  - Logical versus physical file layout
  - File size
  - Examples: asynchronous I/O
- I/O error handling
- I/O error classes
- Examples: double buffering with split collective I/O, subarray filetype constructor

---

## Chapter 5. Programming considerations for user applications in POE

This chapter describes various limitations, restrictions, and programming considerations for user applications written to run under the IBM Parallel Environment for AIX (PE) licensed program, including these topics:

- “The MPI library.”
- “Parallel Operating Environment overview.”
- “POE user limits” on page 26.
- “Exit status” on page 26.
- “POE job step function” on page 27.
- “POE additions to the user executable” on page 27.
- “Threaded programming” on page 36.
- “Using MPI and LAPI in the same program” on page 43.

---

### The MPI library

The MPI library uses hidden AIX kernel threads as well as the users’ threads to move data into and out of message buffers. It supports MPI only, (not MPL, an older IBM proprietary message passing library API), and supports message passing on the main thread and on user-created threads. The MPI library includes support for both 32-bit and 64-bit applications. The hidden threads also ensure that message packets are acknowledged, and when necessary, retransmitted. User applications, when compiled with the PE Version 4 compilation scripts (**mpcc\_r**, **mpCC\_r**, **mpxlf\_r**), will always be compiled with the threaded MPI library, although the application itself may not be threaded.

### The signal library has been removed

In PE Version 4, a single version of the message-passing library is provided. Previous releases provided two versions: a threads library, and a signal-handling library. PE Version 4 provides only a threaded version of the library, with binary compatibility for the signal-handling library functions. In addition, PE Version 4 supports only MPI functions, in both 32-bit and 64-bit applications. MPL is no longer supported.

In addition, the MPI library is using the Low-level communication API (LAPI) protocol as a common transport layer. For more information on this and the use of the LAPI protocol, see *IBM Reliable Scalable Cluster Technology for AIX 5L: LAPI Programming Guide*.

---

### Parallel Operating Environment overview

As the end user, you are encouraged to think of the Parallel Operating Environment (POE) (also referred to as the **poe** command) as an ordinary (serial) command. It accepts redirected I/O, can be run under the **nice** and **time** commands, interprets command flags, and can be invoked in shell scripts.

An  $n$ -task parallel job running in POE consists of: the  $n$  user tasks, a number of instances of the PE partition manager daemon (**pmd**) that is equal to the number of nodes, and the POE home node task in which the **poe** command runs. The **pmd**

is the parent task of the user's task. There is one **pmd** for each node. A **pmd** is started by the POE home node on each machine on which a user task runs, and serves as the point of contact between the home node and the users' tasks.

The POE home node routes standard input, standard output, and standard error streams between the home node and the users' tasks with the **pmd** daemon, using TCP/IP sockets for this purpose. The sockets are created when the POE home node starts the **pmd** daemon for each task of a parallel job. The POE home node and **pmd** also use the sockets to exchange control messages to provide task synchronization, exit status and signaling. These capabilities do not depend on the message passing library, and are available to control any parallel program run by the **poe** command.

---

## POE user limits

When interactive or batch POE applications are submitted under LoadLeveler, it is possible to use the LoadLeveler class to define the user resource limits used for the duration of the job. This also allows LoadLeveler to define and modify a different set of user limits on the submit and compute nodes, using different LoadLeveler job classes.

For interactive POE applications, without using LoadLeveler, POE does not copy or replicate the user resource limits on the remote nodes where the parallel tasks are to run (the compute nodes). POE uses the user limits as defined by the **/etc/security/limits** file. If the user limits on the submitting node (home node) are different than those on the compute nodes, POE does not change the user limits on the compute nodes to match those on the submitting node.

Users should ensure that they have sufficient user resource limits on the compute nodes, when submitting interactive parallel jobs. Users may want to coordinate their user resource needs with their AIX system administrators to ensure that proper user limits are in place, such as in the **/etc/security/limits** file on each node, or by some other means.

---

## Exit status

The exit status is any value from 0 through 255. This value, which is returned from POE on the home node, reflects the composite exit status of your parallel application as follows:

- If **MPI\_ABORT(comm,nn>0,ierror)** or **MPI\_Abort(comm,nn>0)** is called, the exit status is *nn* (mod 256).
- If all tasks terminate using **exit(MM>=0)** or **STOP MM>=0** and **MM** is not equal to 1 and is less than 128 for all nodes, POE provides a synchronization barrier at the exit. The exit status is the largest value of **MM** from any task of the parallel job (mod 256).
- If any task terminates using **exit(MM =1)** or **STOP MM =1**, POE will immediately terminate the parallel job, as if **MPI\_Abort(MPI\_COMM\_WORLD,1)** had been called. This may also occur if an error is detected within a FORTRAN library because a common error response by FORTRAN libraries is to call **STOP 1**.
- If any task terminates with a signal (for example, a segment violation), the exit status is the signal plus 128, and the entire job is immediately terminated.
- If POE terminates before the start of the user's application, the exit status is 1.

- If the user's application cannot be loaded or fails before the user's `main()` is called, the exit status is 255.
- You should explicitly call `exit(MM)` or `STOP MM` to set the desired exit code. A program exiting without an explicit exit value returns unpredictable status, and may result in premature termination of the parallel application and misleading error messages. A well constructed MPI application should terminate with `exit(0)` or `STOP 0` sometime after calling `MPI_FINALIZE`.

---

## POE job step function

The POE job step function is intended for the execution of a sequence of separate yet interrelated dependent programs. Therefore, it provides you with a job control mechanism that allows both job step progression and job step termination. The job control mechanism is the program's exit code.

- Job step progression:  
POE continues the job step sequence if the task exit code is 0 or in the range of 2 through 127.
- Job-step termination:  
POE terminates the parallel job, and does not run any remaining user programs in the job step list if the task exit code is equal to 1 or greater than 127.
- Default termination:  
Any POE infrastructure detected failure (such as failure to open pipes to the child task, or an exec failure to start the user's executable) terminates the parallel job, and does not run any remaining user programs in the job step queue.

---

## POE additions to the user executable

Legacy POE scripts **mpcc**, **mpCC**, and **mpxlf** are now symbolic links to **mpcc\_r**, **mpCC\_r**, and **mpxlf\_r** respectively. The old command names are still used in some of the examples in this book.

POE links in the routines described in the sections that follow, when your executable is compiled with any of the POE compilation scripts, such as: **mpcc\_r**, or **mpxlf\_r**. These topics are discussed:

- "Signal handlers" on page 28.
- "Handling AIX signals" on page 28.
- "Do not hard code file descriptor numbers" on page 29.
- "Termination of a parallel job" on page 29.
- "Do not run your program as root" on page 30.
- "AIX function limitations" on page 30.
- "Shell execution" on page 30.
- "Do not rewind STDIN, STDOUT, or STDERR" on page 30.
- "Do not match blocking and non-blocking collectives" on page 30.
- "Passing string arguments to your program correctly" on page 31.
- "POE argument limits" on page 31.
- "Network tuning considerations" on page 31.
- "Standard I/O requires special attention" on page 32.
- "Reserved environment variables" on page 33.
- "AIX message catalog considerations" on page 33.

- “Language bindings” on page 33.
- “Available virtual memory segments” on page 34.
- “Using a switch clock as a time source” on page 34.
- “Running applications with large numbers of tasks” on page 35.
- “Running POE with MALLOCDDEBUG” on page 35.

## Signal handlers

POE installs signal handlers for most signals that cause program termination, so that it can notify the other tasks of termination. POE then causes the program to exit normally with a code of (*signal* plus 128). This section includes information about installing your own signal handler for synchronous signals.

**Note:** For information about the way POE handles asynchronous signals, see “Handling AIX signals.”

For synchronous signals, you can install your own signal handlers by using the **sigaction()** system call. If you use **sigaction()**, you can use either the *sa\_handler* member or the *sa\_sigaction* member in the **sigaction** structure to define the signal handling function. If you use the *sa\_sigaction* member, the SA\_SIGINFO flag must be set.

For the following signals, POE installs signal handlers that use the **sa\_sigaction** format:

- SIGABRT
- SIGBUS
- SIGEMT
- SIGFPE
- SIGILL
- SIGSEGV
- SIGSYS
- SIGTRAP

POE catches these signals, performs some cleanup, installs the default signal handler (or lightweight core file generation), and re-raises the signal, which will terminate the task.

Users can install their own signal handlers, but they should save the address of the POE signal handler, using a call to SIGACTION. If the user program decides to terminate, it should call the POE signal handler as follows:

```
saved.sa_flags = SA_SIGINFO;
(*saved.sa_sigaction)(signo, NULL, NULL)
```

If the user program decides not to terminate, it should just return to the interrupted code.

**Note:** Do not issue message passing calls, including MPI\_ABORT, from signal handlers. Also, many library calls are not “signal safe”, and should not be issued from signal handlers. See function **sigaction()** in the *AIX Technical Reference* for a list of functions that signal handlers can call.

## Handling AIX signals

The POE runtime environment creates a thread to handle the following asynchronous signals by performing a **sigwait** on them:

- SIGDANGER

- SIGHUP
- SIGINT
- SIGPWR
- SIGQUIT
- SIGTERM

These handlers perform cleanup and exit with a code of (*signal* plus 128). You can install your own signal handler for any or all of these signals. If you want the application to exit after you catch the signal, call the function **pm\_child\_sig\_handler(signal,NULL,NULL)**. The prototype for this function is in file **usr/lpp/ppe.poe/include/pm\_util.h**.

The following asynchronous signals are handled as described below.

### **SIGALRM**

Unlike the now retired signal library, the threads library does not use SIGALRM, and long system calls are not interrupted by the message passing library. For example, **sleep** runs its entire duration unless interrupted by a user-generated event.

### **SIGIO**

Unlike PE 3.2, SIGIO is not used by the MPI library. A user-written signal handler will **not** be called when an MPI packet arrives. The user may use SIGIO for other I/O attention purposes, as required.

### **SIGPIPE**

Some usage environments of the now retired signal library depended on MPI use of SIGPIPE. There is no longer any use of SIGPIPE by the MPI library.

## **Do not hard code file descriptor numbers**

Do not use hard coded file descriptor numbers beyond those specified by STDIN, STDOUT and STDERR.

POE opens several files and uses file descriptors as message passing handles. These are allocated before the user gets control, so the first file descriptor allocated to a user is unpredictable.

## **Termination of a parallel job**

POE provides for orderly termination of a parallel job, so that all tasks terminate at the same time. This is accomplished in the **atexit** routine registered at program initialization. For normal exits (codes 0, and 2 through 127), the **atexit** routine sends a control message to the POE home node, and waits for a positive response. For abnormal exits and those that do not go through the **atexit** routine, the **pmd** daemon catches the exit code and sends a control message to the POE home node.

For normal exits, when POE gets a control message for every task, it responds to each node, allowing that node to exit normally with its individual exit code. The **pmd** daemon monitors the exit code and passes it back to the POE home node for presentation to the user.

For abnormal exits and those detected by **pmd**, POE sends a message to each **pmd** asking that it send a SIGTERM signal to its tasks, thereby terminating the task. When the task finally exits, **pmd** sends its exit code back to the POE home node and exits itself.

User-initiated termination of the POE home node with SIGINT <Ctrl-c> or SIGQUIT <Ctrl-\> causes a message to be sent to **pmd** asking that the appropriate signal be sent to the parallel task. Again, **pmd** waits for the tasks to exit, then terminates itself.

## Do not run your program as root

To prevent uncontrolled root access to the entire parallel job computation resource, POE checks to see that the user is not root as part of its authentication.

## AIX function limitations

Use of the following AIX function may be limited:

- **getuinfo** does not show terminal information, because the user program running in the parallel partition does not have an attached terminal.

## Shell execution

The program executed by POE on the parallel nodes does not run under a shell on those nodes. Redirection and piping of STDIN, STDOUT, and STDERR applies to the POE home node (POE binary), and not the user's code. If shell processing of a command line is desired on the remote nodes, invoke a shell script on the remote nodes to provide the desired preprocessing before the user's application is invoked.

You can have POE run a shell script that is loaded and run on the remote nodes as if it were a binary file.

Due to an AIX limitation, if the program being run by POE is a shell script and there are more than five tasks being run per node, the script **must** be run under **ksh93** by using:

```
#!/bin/ksh93
```

on the first line of the script.

If the POE home node task is not started under the Korn shell, mounted file system names may not be mapped correctly to the names defined for the automount daemon or AIX equivalent. See the *IBM Parallel Environment for AIX: Operation and Use, Volume 1* for a discussion of alternative name mapping techniques.

## Do not rewind STDIN, STDOUT, or STDERR

The partition manager daemon (**pmd**) uses pipes to direct STDIN, STDOUT and STDERR to the user's program. Therefore, do not rewind these files.

## Do not match blocking and non-blocking collectives

The future use of **MPE\_I** non-blocking collectives is deprecated, but only 64-bit executables are affected by this limitation in PE Version 4.

Earlier versions of PE/MPI allowed matching of blocking (MPI) with non-blocking (MPE\_I) collectives. With PE Version 4, it is advised that you do not match blocking and non-blocking collectives in the same collective operation. If you do, a hang situation can occur. It is possible that some existing applications may hang, when run using PE Version 4. In the case of an unexpected hang, turn on DEVELOP mode by setting the environment variable **MP\_EUIDEVELOP** to **yes**, and rerun your application. DEVELOP mode will detect and report any mismatch.

If DEVELOP mode identifies a mismatch, you may continue to use the application as is, by setting **MP\_SHARED\_MEMORY** to **no**. If possible, alter the application to remove the matching of non-blocking with blocking collectives.

## Passing string arguments to your program correctly

Quotation marks, either single or double, used as argument delimiters are stripped away by the shell and are never seen by **poe**. Therefore, the quotation marks must be escaped to allow the quoted string to be passed correctly to the remote tasks as one argument. For example, if you want to pass the following string to the user program (including the embedded blank)

```
a b
```

you need to enter the following:

```
poe user_program \"a b\"
```

**user\_program** is passed the following argument as one token:

```
a b
```

Without the backslashes, the string would have been treated as two arguments (*a and b*).

POE behaves like **rsh** when arguments are passed to POE. Therefore, this command:

```
poe user_program "a b"
```

is equivalent to:

```
rsh some_machine user_program "a b"
```

In order to pass the string argument as one token, the quotation marks have to be escaped using the backslash.

## POE argument limits

The maximum length for POE program arguments is 24576 bytes. This is a fixed limit and cannot be changed. If this limit is exceeded, an error message is displayed and POE terminates. The length of the remote program arguments that can be passed on POE's command line is 24576 bytes minus the number of bytes that are used for POE arguments.

## Network tuning considerations

Programs generating large volumes of STDOUT or STDERR may overload the home node. As described previously, STDOUT and STDERR files generated by a user's program are piped to **pmd**, then forwarded to the POE binary using a TCP/IP socket. It is possible to generate so much data that the IP message buffers on the home node are exhausted, the POE binary hangs, and possibly the entire node hangs. Note that the option **-stdoutmode** (environment variable **MP\_STDOUTMODE**) controls which output stream is displayed by the POE binary, but does not limit the standard output traffic received from the remote nodes, even when set to display the output of only one node.

The POE environment variable **MP\_SNDBUF** can be used to override the default network settings for the size of the TCP/IP buffers used.

If you have large volumes of standard input or output, work with your network administrator to establish appropriate TCP/IP tuning parameters. You may also want to investigate whether using named pipes is appropriate for your application.

## Standard I/O requires special attention

When your program runs on the remote nodes, it has no controlling terminal. STDIN, STDOUT, and STDERR are always piped.

Running the **poe** command (or starting a program compiled with one of the POE compile scripts) causes POE to perform this sequence of events:

1. The POE binary is loaded on the machine on which you submitted the command (the POE home node).
2. The POE binary, in turn, starts a partition manager daemon (**pmd**) on each parallel node assigned to run the job, and tells that **pmd** to run one or more copies of your executable (using **fork** and **exec**).
3. The POE binary reads STDIN and passes it to each **pmd** with a TCP/IP socket connection.
4. The **pmd** on each node pipes STDIN to the parallel tasks on that node.
5. STDOUT and STDERR from the tasks are piped to the **pmd** daemon.
6. This output is sent by the **pmd** on the TCP/IP socket back to the home node POE.
7. This output is written to the POE binary's STDOUT and STDERR descriptors.

Programs that depend on piping standard input or standard output as part of a processing sequence may wish to bypass the home node POE binary. If you know that the task reading STDIN or writing STDOUT must be on the same node (processor) as the POE binary (the POE home node), named pipes can be used to bypass POE's reading and forwarding STDIN and STDOUT.

If your MPI program processes STDIN from a large file on the home node, you must do one of the following:

- Invoke MPI\_INIT before performing any STDIN processing.
- Ensure that all STDIN has been processed (EOF) before invoking MPI\_INIT (or if LAPI is being used in the application, LAPI\_INIT).

If STDIN is piped (or redirected) to the POE binary (with ordinary pipes), handle STDIN in the following way:

- If all of STDIN is read by your program before MPI\_INIT is called, set the environment variable **MP\_HOLD\_STDIN** to **NO**.
- If none of STDIN is read before MPI\_INIT is called, set the environment variable **MP\_HOLD\_STDIN** to **YES**.
- If STDIN is less than approximately 4000 bytes in length, set **MP\_HOLD\_STDIN** to **NO**.
- If none of the above applies, it may not be possible to run your program correctly, and you will have to devise some other mechanism for providing data to your program.

## STDIN and STDOUT piping example

The following two scripts show how STDIN and STDOUT can be piped directly between preprocessing and postprocessing steps, bypassing the POE home node task. This example assumes that parallel task 0 is known or forced to be on the same node as the POE home node.

The script **compute\_home** runs on the home node; the script **compute\_parallel** runs on the parallel nodes (those running tasks 0 through  $n-1$ ).

```
compute_home:
#!/bin/ksh93
# Example script compute_home runs three tasks:
#   data_generator creates/gets data and writes to stdout
#   data_processor is a parallel program that reads data
#   from stdin, processes it in parallel, and writes
#   the results to stdout.
#   data_consumer reads data from stdin and summarizes it
#
mkfifo poe_in_$$
mkfifo poe_out_$$
export MP_STDOUTMODE=0
export MP_STDINMODE=0
data_generator >poe_in_$$ |
    poe compute_parallel poe_in_$$ poe_out_$$ data_processor |
    data_consumer <poe_out_$$
rc=$?
rm poe_in_$$
rm poe_out_$$
exit rc

compute_parallel:
#!/bin/ksh93
# Example script compute_parallel is a shell script that
#   takes the following arguments:
#   1) name of input named pipe (stdin)
#   2) name of output named pipe (stdout)
#   3) name of program to be run (and arguments)
#
poe_in=$1
poe_out=$2
shift 2
$* <$poe_in >$poe_out
```

## Reserved environment variables

Environment variables whose name begins with **MP\_** are intended for use by POE, and should be set only as instructed in the documentation. POE also uses a handful of **MP\_** environment variables for internal purposes, which should not be interfered with.

If the value of **MP\_INFOLEVEL** is greater than or equal to **1**, POE will display any **MP\_** environment variables that it does not recognize, but POE will continue working normally.

## AIX message catalog considerations

POE assumes that the environment variable **NLSPATH** contains the appropriate POE message catalogs, even if environment variable **LANG** is set to **C** or is not set. Duplicate message catalogs are provided for languages **En\_US**, **en\_US**, and **C**.

## Language bindings

The FORTRAN, C, and C++ bindings for MPI are contained in the same library and can be freely intermixed. The library is named **libmpi\_r.a**. Because it contains both 32-bit and 64-bit objects, and the compiler and linker select between them, **libmpi\_r.a** can be used for both 32-bit and 64-bit applications.

The AIX compilers support the flag **-qarch**. This option allows you to target code generation for your application to a particular processor architecture. While this option can provide performance enhancements on specific platforms, it inhibits

portability. The MPI library is not targeted to a specific architecture, and is not affected by the flag **-qarch** on your compilation.

The MPI standard includes several routines that take *choice* arguments. For example MPI\_SEND may be passed a buffer of REAL on one call, and a buffer of INTEGER on the next. The **-qextcheck** compiler option flags this as an error. In F77, *choice* arguments are a violation of the FORTRAN standard that few compilers would complain about. In F90, *choice* arguments can be interpreted by the compiler as an attempt to use function overloading. MPI FORTRAN functions do not require genuine overloading support to give correct results and PE MPI does not define overloaded functions for all potential *choice* arguments. Because **-qextcheck** considers use of *choice* arguments to be erroneous overloads even though the code is correct MPI, the **-qextcheck** option should not be used.

## Available virtual memory segments

A 32-bit application is limited to 16 segments. The AIX memory model for 32-bit applications claims five of these. The application can allocate up to eight segments (2 GB) for application data (the heap, specified with compile option **-bmaxdata**). The communication subsystem takes a variable number of segments, depending on options chosen at run time. In some circumstances, for 32-bit applications the total demand for segments can be greater than 16 and a job will be unable to start or will run with reduced performance. If your application is using a very large heap and you consider enabling striping, see the migration section in *IBM Parallel Environment for AIX 5L: Operation and Use, Volume 1* for details.

## Using a switch clock as a time source

The high performance switch interconnects that supports user space also provide a globally-synchronized counter that can be used as a source for the MPI\_WTIME function, provided that all tasks are run on nodes connected to the same switch interconnect. The environment variable **MP\_CLOCK\_SOURCE** provides additional control.

Table 3 on page 35 shows how the clock source is determined. PE MPI guarantees that the MPI attribute MPI\_WTIME\_IS\_GLOBAL has the same value at every task, and all tasks use the same clock source (AIX or switch).

Table 3. How the clock source is determined

MP_CLOCK_SOURCE	Library version	Are all nodes on the same switch?	Source used	MPI_WTIME_IS_GLOBAL
AIX	ip	yes	AIX	false
AIX	ip	no	AIX	false
AIX	us	yes	AIX	false
AIX	us	no	Error	false
SWITCH	ip	yes*	switch	true
SWITCH	ip	no	AIX	false
SWITCH	us	yes	switch	true
SWITCH	us	no	Error	
not set	ip	yes	switch	false
not set	ip	no	AIX	false
not set	us	yes	switch	true
not set	us	no	Error	
<b>Note:</b> * If <b>MPI_WTIME_IS_GLOBAL</b> value is to be trusted, the user is responsible for making sure all of the nodes are connected to the <b>same</b> switch. If the job is in IP mode and <b>MP_CLOCK_SOURCE</b> is left to default, <b>MPI_WTIME_IS_GLOBAL</b> will report false even if the switch is used because MPI cannot know it is the same switch.				
In this table, ip refers to IP protocol, us refers to User Space protocol.				

## Running applications with large numbers of tasks

If you plan to run your parallel applications with a large number of tasks (more than 256), the following tips may improve stability and performance:

- To control the amount of memory made available for early arrival buffering, the environment variable **MP\_BUFFER\_MEM** or command-line flag **-buffer\_mem** can accept the format *M1, M2* where each of *M1, M2* is a memory specification suffixed with K, M, or G.  
*M1* specifies the amount of pre-allocated memory. *M2* specifies the maximum memory that might be requested by the program. See the entry for **MP\_BUFFER\_MEM** in Chapter 11, “POE environment variables and command-line flags,” on page 69 and Appendix E, “PE MPI buffer management for eager protocol,” on page 217 for details.
- When using IP mode, use a host list file with the switch IP names, instead of the IP host name.
- In 32-bit applications, you may avoid the problem of running out of memory by linking applications with an extended heap starting with data segment 3. For example, specifying the **-bD:0x30000000** loader option causes segments 3, 4, and 5 to be allocated to the heap. The default is to share data segment 2 between the stack and the heap.

For limitations on the number of tasks, tasks per node, and other restrictions, see Chapter 10, “MPI size limits,” on page 65.

## Running POE with MALLOCDEBUG

Running a POE job that uses MALLOCDEBUG with an align:n option of other than 8 may result in undefined behavior. To allow the parallel program being run

by POE (**myprog**, for example) to run with an `align:n` option of other than 8, create the following script (called **myprog.sh**), for example:

```
MALLOCTYPE=debug
MALLOCDDEBUG=align:0
myprog myprog_options
```

and then run with this command:

```
poe myprog.sh poe_options
```

instead of this command:

```
poe myprog poe_options myprog_options
```

---

## Threaded programming

When programming in a threads environment, specific skills and considerations are required. The information in this subsection provides you with specific programming considerations when using POE and the MPI library. This section assumes that you are familiar with POSIX threads in general, including multiple execution threads, thread condition waiting, thread-specific storage, thread creation and thread termination. These topics are discussed:

- “Running single threaded applications.”
- “POE gets control first and handles task initialization” on page 37.
- “Limitations in setting the thread stack size” on page 37.
- “Forks are limited” on page 37.
- “Thread-safe libraries” on page 37.
- “Program and thread termination” on page 37.
- “Order requirement for system includes” on page 38.
- “Using MPI\_INIT or MPI\_INIT\_THREAD” on page 38.
- “Collective communication calls” on page 38.
- “Support for M:N threads” on page 38.
- “Checkpoint and restart limitations” on page 39.
- “64-bit application considerations” on page 42.
- “MPI\_WAIT\_MODE: the nopoll option” on page 43.
- “Mixed parallelism with MPI and threads” on page 43.

## Running single threaded applications

As mentioned earlier, PE Version 4 provides only the threaded version of the MPI library and program compiler scripts.

Applications that do not intend to use threads can continue to run as single threaded programs, despite the fact they are now compiled as threaded programs. However there are some side issues application developers should be aware of. Any application that was compiled with the signal library compiler scripts prior to PE Version 4 and not using MPE\_I non-blocking collectives, is in this class.

Application performance may be impacted by locking overheads in the threaded MPI library. Users with applications that do not create additional threads and do not use the nonstandard MPE\_I nonblocking collectives, MPI-IO, or MPI one-sided communication may wish to set the environment variable **MP\_SINGLE\_THREAD** to **yes** for a possible performance improvement.

Do not set `MP_SINGLE_THREAD` to **yes** unless you are **certain** that the application is single threaded. Setting `MP_SINGLE_THREAD` to **yes**, and then creating additional user threads will give unpredictable results. Calling `MPI_FILE_OPEN`, `MPI_WIN_CREATE` or any MPE\_I nonblocking collective in an application running with `MP_SINGLE_THREAD` set to **yes** will cause PE MPI to terminate the job.

Also, applications that register signal handlers may need to be aware that the execution is in a threaded environment.

## POE gets control first and handles task initialization

POE sets up its environment using the `poe_remote_main` entry point. The `poe_remote_main` entry point sets up signal handlers, initializes a thread for handling asynchronous communication, and sets up an `atexit` routine before your main program is invoked. MPI communication is established when you call `MPI_INIT` in your application, and not during `poe_remote_main`.

## Limitations in setting the thread stack size

The main thread stack size is the same as the stack size used for non-threaded applications. Library-created service threads use a default stack size of 8K for 32-bit applications and 16K for 64-bit applications. The default value is specified by the variable `PTHREAD_STACK_MIN`, which is defined in header file `/usr/include/limits.h`.

If you write your own MPI reduction functions to use with nonblocking collective communications, these functions may run on a service thread. If your reduction functions require significant amounts of stack space, you can use the `MP_THREAD_STACKSIZE` environment variable to cause larger stacks to be created for service threads. This does not affect the default stack size for any threads you create.

## Forks are limited

If a task forks, only the thread that forked exists in the child task. Therefore, the message passing library will not operate properly. Also, if the forked child does not exec another program, it should be aware that an `atexit` routine has been registered for the parent that is also inherited by the child. In most cases, the `atexit` routine requests that POE terminate the task (parent). A forked child should terminate with an `_exit(0)` system call to prevent the `atexit` routine from being called. Also, if the forked parent terminates before the child, the child task will not be cleaned up by POE.

**Note:** A forked child must **not** call the message passing library (MPI library).

## Thread-safe libraries

Most AIX libraries are thread-safe, such as `libc.a`. However, not all libraries have a thread-safe version. It is your responsibility to determine whether the AIX libraries you use can be safely called by more than one thread.

## Program and thread termination

`MPI_FINALIZE` terminates the MPI service threads but does not affect user-created threads. Use `pthread_exit` to terminate any user-created threads, and `exit(m)` to terminate the main program (initial thread). The value of *m* is used to set POE's

exit status as explained in “Exit status” on page 26. For programs that are successful, the value for *m* should be zero.

## Order requirement for system includes

For programs that explicitly use threads, AIX requires that the system include file **pthread.h** must be first, with **stdio.h** or other system includes following it. **pthread.h** defines some conditional compile variables that modify the code generation of subsequent includes, particularly **stdio.h**. Note that **pthread.h** is not required unless your program uses thread-related calls or data.

## Using MPI\_INIT or MPI\_INIT\_THREAD

Call MPI\_INIT once per task, not once per thread. MPI\_INIT does not have to be called on the main thread, but MPI\_INIT and MPI\_FINALIZE must be called on the same thread.

MPI calls on other threads must adhere to the MPI standard in regard to the following:

- A thread cannot make MPI calls until MPI\_INIT has been called.
- A thread cannot make MPI calls after MPI\_FINALIZE has been called.
- Unless there is a specific thread synchronization protocol provided by the application itself, you cannot rely on any specific order or speed of thread processing.

The MPI\_INIT\_THREAD call allows the user to request a level of thread support ranging from MPI\_THREAD\_SINGLE to MPI\_THREAD\_MULTIPLE. PE MPI ignores the request argument. If **MP\_SINGLE\_THREAD** is set to **yes**, MPI runs in a mode equivalent to MPI\_THREAD\_FUNNELED. If **MP\_SINGLE\_THREAD** is set to **no**, or allowed to default, PE MPI runs in MPI\_THREAD\_MULTIPLE mode.

The nonstandard MPE\_I nonblocking collectives, MPI-IO, and MPI one-sided communication will not operate if **MP\_SINGLE\_THREAD** is set to **yes**.

## Collective communication calls

Collective communication calls must meet the MPI standard requirement that all participating tasks execute collective communication calls on any given communicator in the same order. If collective communications call are made on multiple threads, it is your responsibility to ensure the proper sequencing. The preferred approach is for each thread to use a distinct communicator.

## Support for M:N threads

By default, AIX causes thread creation to use process scope. POE overrides this default by setting the environment variable **AIXTHREAD\_SCOPE** to **S**, which has the effect that all user threads are created with system contention scope, with each user thread mapped to a kernel thread. If you explicitly set **AIXTHREAD\_SCOPE** to **P**, to be able to create to your user threads with process contention scope, POE will not override your setting. In process scope, **M** number of user threads are mapped to **N** number of kernel threads. The values of the ratio **M:N** can be set by an AIX environment variable.

The service threads created by MPI, POE, and LAPI have system contention scope, that is, they are mapped 1:1 to kernel threads.

Any user-created thread that began with process contention scope, will be converted to system contention scope when it makes its first MPI call. Threads that must remain in process contention scope should not make MPI calls.

## Checkpoint and restart limitations

Use of the checkpoint and restart function has these limitations:

- “Programs that cannot be checkpointed.”
- “Program restrictions.”
- “AIX function restrictions” on page 40.
- “Node restrictions” on page 40.
- “Task-related restrictions” on page 41.
- “Pthread and atomic lock restrictions” on page 41.
- “Other restrictions” on page 41.

### Programs that cannot be checkpointed

The following programs cannot be checkpointed:

- Programs that do not have the environment variable **CHECKPOINT** set to **yes**.
- Programs that are being run under:
  - The dynamic probe class library (DPCL).
  - Any debugger that is not checkpoint/restart-capable.
- Processes that use:
  - Extended **shmat** support
  - Pinned shared memory segments
- Sets of processes in which any process is running a **setuid** program when a checkpoint occurs.
- Jobs for which POE input or output is a pipe.
- Jobs for which POE input or output is redirected, unless the job is submitted from a shell that had the **CHECKPOINT** environment variable set to **yes** before the shell was started. If POE is run from inside a shell script and is run in the background, the script must be started from a shell started in the same manner for the job to be able to be checkpointed.
- Jobs that are run using the switch or network table sample programs.
- Interactive POE jobs for which the **su** command was used prior to checkpointing or restarting the job.

### Program restrictions

Any program that meets both these criteria:

- is compiled with one of the threaded compile scripts provided by PE
- may be checkpointed prior to its **main()** function being invoked

must wait for the **0031-114** message to appear in POE’s **STDERR** before issuing the checkpoint of the parallel job. Otherwise, a subsequent restart of the job may fail.

**Note:** The **MP\_INFOLEVEL** environment variable, or the **-infolevel** command-line option, must be set to a value of at least 2 for this message to appear.

Any program that meets both these criteria:

- is compiled with one of the threaded compile scripts provided by PE
- may be checkpointed immediately after the parallel job is restarted

must wait for the **0031-117** message to appear in POE's STDERR before issuing the checkpoint of the restarted job. Otherwise, the checkpoint of the job may fail.

**Note:** The **MP\_INFOLEVEL** environment variable, or the **-infolevel** command line option, must be set to a value of at least 2 for this message to appear.

## AIX function restrictions

The following AIX functions will fail, with an errno of ENOTSUP, if the **CHECKPOINT** environment variable is set to **yes** in the environment of the calling program:

clock_getcpuclockid()	pthread_mutex_getprioceiling()
clock_getres()	pthread_mutex_setprioceiling()
clock_gettime()	pthread_mutex_timedlock()
clock_nanosleep()	pthread_rwlock_timedrdlock()
clock_settime()	pthread_rwlock_timedwrlock()
mlock()	pthread_setschedprio()
mlockall()	pthread_spin_destroy()
mq_close()	pthread_spin_init()
mq_getattr()	pthread_spin_lock()
mq_notify()	pthread_spin_trylock()
mq_open()	pthread_spin_unlock()
mq_receive()	sched_getparam()
mq_send()	sched_get_priority_max()
mq_setattr()	sched_get_priority_min()
mq_timedreceive()	sched_getscheduler()
mq_timedsend()	sched_rr_get_interval()
mq_unlink()	sched_setparam()
munlock()	sched_setscheduler()
munlockall()	sem_close()
nanosleep()	sem_destroy()
pthread_barrierattr_init()	sem_getvalue()
pthread_barrierattr_destroy()	sem_init()
pthread_barrierattr_getpshared()	sem_open()
pthread_barrierattr_setpshared()	sem_post()
pthread_barrier_destroy()	sem_timedwait()
pthread_barrier_init()	sem_trywait()
pthread_barrier_wait()	sem_unlink()
pthread_condattr_getclock()	sem_wait()
pthread_condattr_setclock()	shm_open()
pthread_getcpuclockid()	shm_unlink()
pthread_mutexattr_getprioceiling()	timer_create()
pthread_mutexattr_getprotocol()	timer_delete()
pthread_mutexattr_setprioceiling()	timer_getoverrun()
pthread_mutexattr_setprotocol()	timer_gettime()
	timer_settime()

## Node restrictions

The node on which a process is restarted must have:

- The same operating system level (including PTFs). In addition, a restarted process may not load a module that requires a system call from a kernel extension that was not present at checkpoint time.
- The same switch type as the node where the checkpoint occurred.

If any threads in the parallel task were bound to a specific processor ID at checkpoint time, that processor ID must exist on the node where that task is restarted.

## Task-related restrictions

- The number of tasks and the task geometry (the tasks that are common within a node) must be the same on a restart as it was when the job was checkpointed.
- Any regular file open in a parallel task when that task is checkpointed must be present on the node where that task is restarted, including the executable and any dynamically loaded libraries or objects.
- If any task within a parallel application uses sockets or pipes, user callbacks should be registered to save data that may be in transit when a checkpoint occurs, and to restore the data when the task is resumed after a checkpoint or restart. Similarly, any user shared memory should be saved and restored.

## Pthread and atomic lock restrictions

- A checkpoint operation will not begin on a parallel task until each user thread in that task has released all pthread locks, if held.

This can potentially cause a significant delay from the time a checkpoint is issued until the checkpoint actually occurs. Also, any thread of a process that is being checkpointed that does not hold any pthread locks and tries to acquire one will be stopped immediately. There are no similar actions performed for atomic locks (`_check_lock` and `_clear_lock`, for example).

- Atomic locks must be used in such a way that they do not prevent the releasing of pthread locks during a checkpoint.

For example, if a checkpoint occurs and thread 1 holds a pthread lock and is waiting for an atomic lock, and thread 2 tries to acquire a different pthread lock (and does not hold any other pthread locks) before releasing the atomic lock that thread 1 is waiting for, the checkpoint will hang.

- If a pthread lock is held when a parallel task creates a new process (either implicitly using `popen`, for example, or explicitly using `fork` or `exec`) and the releasing of the lock is contingent on some action of the new process, the **CHECKPOINT** environment variable must be set to **no** before causing the new process to be created.

Otherwise, the parent process may be checkpointed (but not yet stopped) before the creation of the new process, which would result in the new process being checkpointed and stopped immediately.

- A parallel task must not hold a pthread lock when creating a new process (either implicitly using `popen` for example, or explicitly using `fork`) if the releasing of the lock is contingent on some action of the new process.

Otherwise a checkpoint could occur that would cause the child process to be stopped before the parent could release the pthread lock causing the checkpoint operation to hang.

- The checkpoint operation may hang if any user pthread locks are held across:
  - Any collective communication calls in MPI (or if LAPI is being used in the application, LAPI).
  - Calls to `mpc_init_ckpt` or `mp_init_ckpt`.
  - Any blocking MPI call that returns only after action on some other task.

## Other restrictions

- Processes cannot be profiled at the time a checkpoint is taken.
- There can be no devices other than TTYs or `/dev/null` open at the time a checkpoint is taken.
- Open files must either have an absolute pathname that is less than or equal to `PATHMAX` in length, or must have a relative pathname that is less than or equal

to PATHMAX in length from the current directory at the time they were opened. The current directory must have an absolute pathname that is less than or equal to PATHMAX in length.

- Semaphores or message queues that are used within the set of processes being checkpointed must only be used by processes within the set of processes being checkpointed.

This condition is not verified when a set of processes is checkpointed. The checkpoint and restart operations will succeed, but inconsistent results can occur after the restart.

- The processes that create shared memory must be checkpointed with the processes using the shared memory if the shared memory is ever detached from all processes being checkpointed. Otherwise, the shared memory may not be available after a restart operation.
- The ability to checkpoint and restart a process is not supported for B1 and C2 security configurations.
- A process can checkpoint another process only if it can send a signal to the process.

In other words, the privilege checking for checkpointing processes is identical to the privilege checking for sending a signal to the process. A privileged process (the effective user ID is 0) can checkpoint any process. A set of processes can only be checkpointed if each process in the set can be checkpointed.

- A process can restart another process only if it can change its entire privilege state (real, saved, and effective versions of user ID, group ID, and group list) to match that of the restarted process.
- A set of processes can be restarted only if each process in the set can be restarted.

## 64-bit application considerations

Support for 64-bit applications is provided in the MPI library. You can choose 64-bit support by specifying **-q64** as a compiler flag, or by setting the environment variable **OBJECT\_MODE** to **64** at compile and link time. All objects in a 64-bit environment must be compiled with **-q64**. You cannot call a 32-bit library from a 64-bit application, nor can you call a 64-bit library from a 32-bit application.

Integers passed to the MPI library are always 32 bits long. If you use the FORTRAN compiler directive **-qintsize=8** as your default integer length, you will need to type your MPI integer arguments as **INTEGER\*4**. All integer parameters in **mpif.h** are explicitly declared **INTEGER\*4** to prevent **-qintsize=8** from altering their length.

As defined by the MPI standard, the *count* argument in MPI send and receive calls is a default size signed integer. In AIX, even 64-bit executables use 32-bit integers by default. To send or receive extremely large messages, you may need to construct your own datatype (for example, a 'page' datatype of 4096 contiguous bytes).

The FORTRAN compilation scripts **mpxlf\_r**, **mpxlf90\_r**, and **mpxlf95\_r** set the include path for **mpif.h** to: **/usr/lpp/ppe.poe/include/thread64** or **/usr/lpp/ppe.poe/include/thread**, as appropriate. Do not add a separate include path to **mpif.h** in your compiler scripts or make files, as an incorrect version of **mpif.h** could be picked up in compilation, resulting in subtle run time errors.

The AIX 64-bit address space is large enough to remove any limitations on the number of memory segments that can be used, so the information in “Available virtual memory segments” on page 34 does not apply to the 64-bit library.

## **MPI\_WAIT\_MODE: the nopoll option**

Environment variable **MPI\_WAIT\_MODE** set to **nopoll** is supported as an option. It causes a blocking MPI call to go into a system wait after approximately one millisecond of polling without a message being received. **MPI\_WAIT\_MODE** set to **nopoll** may reduce CPU consumption for applications that post a receive call on a separate thread, and that receive call does not expect an immediate message arrival. Also, using **MPI\_WAIT\_MODE** set to **nopoll** may increase delay between message arrival and the blocking call’s return. It is recommended that **MP\_CSS\_INTERRUPT** be set to **yes** when the **nopoll** wait is selected, so that the system wait can be interrupted by the arrival of a message. Otherwise, the **nopoll** wait is interrupted at the timing interval set by **MP\_POLLING\_INTERVAL**.

## **Mixed parallelism with MPI and threads**

The MPI programming model provides parallelism by using multiple tasks that communicate by making message passing calls. Many of these MPI calls can block until some action occurs on another task. Examples include collective communication, collective **MPI-IO**, **MPI\_SEND**, **MPI\_RECV**, **MPI\_WAIT**, and the synchronizations for MPI one-sided.

The threads model provides parallelism by running multiple execution streams in a single address space, and can depend on data object protection or order enforcement by mutex lock. Threads waiting for a mutex are blocked until the thread holding the mutex releases it. The thread holding the mutex will not release it until it completes whatever action it took the lock to protect. If you choose to do mutex lock protected threads parallelism and MPI task parallelism in a single application, you must be careful not to create interlocks between blocking by MPI call and blocking on mutex locks. The most obvious rule is: avoid making a blocking MPI call while holding a mutex.

OpenMP and MPI in a single application offers relative safety because the OpenMP model normally involves distinct parallel sections in which several threads are spawned at the beginning of the section and joined at the end. The communication calls occur on the main thread and outside of any parallel section, so they do not require mutex protection. This segregation of threaded epochs from communication epochs is safe and simple, whether you use OpenMP or provide your own threads parallelism.

The threads parallelism model in which some number of threads proceed in a more or less independent way, but protect critical sections (periods of protected access to a shared data object) with locks requires more care. In this model, there is much more chance you will hold a lock while doing a blocking MPI operation related to some shared data object.

---

## **Using MPI and LAPI in the same program**

You can use MPI and LAPI concurrently in the same parallel program. Their operation is logically independent of one another, and you can specify independently whether each uses the User Space protocol or the IP protocol.

If both MPI and LAPI use the same protocol (either User Space or IP), you can choose to have them share the underlying packet protocol (User Space or UDP).

You do this by setting the POE environment variable **MP\_MSG\_API** to **mpi\_lapi**. If you do not wish to share the underlying packet protocol, set **MP\_MSG\_API** to **mpi,lapi**.

In User Space, running with shared resource **MP\_MSG\_API** set to **mpi\_lapi** causes LoadLeveler to allocate only one window for the MPI/LAPI pair, rather than two windows. Since each window takes program resources (segment registers, memory for DMA send and receive FIFOs, adapter buffers and network tables), sharing the window makes sense if MPI and LAPI are communicating at different times (during different phases of the program). If MPI and LAPI are doing concurrent communication, the DMA receive buffer may be too small to contain packets from both LAPI and MPI, and packets may be dropped. This may impair performance.

The **MP\_CSS\_INTERRUPT** environment variable applies only to the MPI API. At MPI\_INIT time, MPI sets the protocol for the LAPI instance that MPI is using, according to MPI defaults or as indicated by environment variable **MP\_CSS\_INTERRUPT**. In non-shared mode, MPI retains control of the LAPI instance that it is using. If there is use of the LAPI API in the same application, the LAPI\_Senv() function can be used to control interrupts for the LAPI API instance, without affecting the instance that MPI is using.

In shared mode, MPI\_INIT sets interrupt behavior of its LAPI instance, just as in non-shared mode, but MPI has no way to recognize or control changes to the interrupt mode of this shared instance that may occur later through the LAPI\_Senv() function. Unexpected changes in interrupt mode made with the LAPI API to the LAPI instance being shared with MPI can affect MPI performance, but will not affect whether a valid MPI program runs correctly.

In IP, running with shared resource **MP\_MSG\_API** set to **mpi\_lapi** uses only one pair of UDP ports, while running with separated resource **MP\_MSG\_API** set to **mpi,lapi** uses two pair of UDP ports. In the separated case, there may be a slight increase in job startup time due to the need for POE to communicate two sets of port lists.

## Differences between MPI in PE 3.2 and PE Version 4

PE 3.2 MPI used an underlying transport layer called MPCl, which was an internal component of PSSP for the RS/6000 SP system (Cluster 1600). MPCl provided a **reliable byte stream** interface, in which user's data is copied to a **send pipe** whose size is up to 64 KB, which is then broken into packets and sent to the receiver. The receiver assembles the received packets into a **receive pipe**, and populates the user's data reads from the receive pipe. For programs with large numbers of tasks, the amount of memory allocated to pipes becomes quite large, and reduces the amount of storage available for user data.

In PE Version 4, MPI uses an underlying transport called LAPI, which is distributed as an AIX fileset, part of the RSCT component. In contrast to the **reliable byte stream** approach of MPCl, LAPI provides a **reliable message** protocol, which uses much less storage for jobs with a large number of tasks.

Because the underlying transport mechanism is so different, POE MPI environment variables used to tune MPCl performance are, in some cases, ignored. Also, there are new environment variables to tune the LAPI operation. The following variables, and their corresponding command-line options, are now ignored:

- **MP\_INTRDELAY**, and the corresponding function **mp\_intrdelay**
- **MP\_SYNC\_ON\_CONNECT**

- **MP\_PIPE\_SIZE**
- **MP\_ACK\_INTERVAL**

The following variables are new. A brief description of their intended function is provided. For more details, see Chapter 11, “POE environment variables and command-line flags,” on page 69.

**MP\_UDP\_PACKET\_SIZE**

Specifies the UDP datagram size to be used for UDP/IP message transport.

**MP\_ACK\_THRESH**

Sets the threshold for return packet flow control acknowledgements.

**MP\_USE\_BULK\_XFER**

Causes the use of the Remote Direct Memory Access (RDMA) capability. See “Remote Direct Memory Access (RDMA) considerations” on page 9.

## Differences between MPI in PE 4.1 and PE 4.2

- Environment variable **MP\_SHARED\_MEMORY** now has a default of **yes**.
- Environment variable **MP\_BUFFER\_MEM** has been enhanced. See Chapter 11, “POE environment variables and command-line flags,” on page 69.
- Refer to “Summary of changes for Parallel Environment 4.2” on page xii for other differences.

## Other differences

- Handling shared memory. See Chapter 3, “Using shared memory,” on page 15.
- The MPI communication subsystem is activated at **MPI\_INIT** and closed at **MPI\_FINALIZE**. When MPI and LAPI share the subsystem, whichever call comes first between **MPI\_INIT** and **LAPI\_INIT** will provide the activation. Whichever call comes last between **MPI\_FINALIZE** and **LAPI\_TERM** will close it.
- Additional service threads. See “POE-supplied threads.”

## POE-supplied threads

Your parallel program is normally run under the control of POE. The communication stack includes MPI, LAPI, and the hardware interface layer. The communication stack also provides access to the global switch clock. This stack makes use of several internally spawned threads. The options under which the job is run affect which threads are created, therefore some, but not all, of the threads listed below are created in a typical application run. Most of these threads sleep in the kernel waiting for notification of some rare condition and do not compete for CPU access during normal job processing. When a job is run in polling mode, there will normally be little CPU demand by threads other than the users’ application threads.

There can be MPI service threads spawned to handle **MPE\_I** non-blocking collective communication, **MPI-IO**, and **MPI** one-sided communication. The threads are spawned as needed and kept for reuse. An application that uses none of these functions will not have any of these threads. An application that uses **MPE\_I** non-blocking collective communication, **MPI-IO**, or **MPI** one-sided communication will spawn one or more MPI service threads at first need. When the operation that required the thread finishes, the thread will be left sleeping in the kernel and will be visible in the debugger. At a subsequent need, if a sleeping thread is available, it is triggered for reuse to carry out the non-blocking collective communication **MPI-IO** and **MPI** one-sided operation. While waiting to be reused, the threads do

not consume significant resources. The MPE\_I, MPI-IO, or MPI one-sided API call that triggers one of these service threads to run in a given task can, and usually does, come from some remote task. There can be substantial CPU usage by these threads when non-blocking collective communication MPI-IO, or MPI one-sided communication is active.

This information is provided to help you understand what you will see in a debugger when examining an MPI task. You can almost always ignore the service threads in your debugging but you may need to find your own thread before you can understand your application behavior. The **dbx** commands **threads** and **thread current n** are useful for displaying the threads list and switching focus to the thread you need to debug.

Table 4 is an example POE/MPI/LAPI thread inventory, in order of thread creation. The list assumes shared memory over two windows, MPI only. Simpler environments (depending on options selected) will involve fewer threads.

Table 4. POE/MPI/LAPI Thread Inventory

Name	Description
T1	User's main program
T2	POE asynchronous exit thread (SIGQUIT, SIGTERM, and so forth)
T3	hardware interface layer device interrupt/timer thread
T4	hardware interface layer fault service handler thread
T5	LAPI Completion handler thread (one default)
T6	LAPI Shared memory dispatcher (shared memory only)
T7	Switch clock service thread
T8	MPI Service threads (if MPE_I nonblocking collective communication, MPI-IO or MPI one-sided calls used). As many as eight are created as required.

---

## Chapter 6. Using error handlers

This chapter provides information on using error handlers.

---

### Predefined error handler for C++

The C++ language interface for MPI includes the predefined error handler `MPI::ERRORS_THROW_EXCEPTIONS` for use with `MPI::Comm::Set_errhandler`, `MPI::File::Set_errhandler`, and `MPI::Win::Set_errhandler`.

`MPI::ERRORS_THROW_EXCEPTIONS` can be set or retrieved only by C++ functions. If a non-C++ program causes an error that invokes the `MPI::ERRORS_THROW_EXCEPTIONS` error handler, the exception will pass up the calling stack until C++ code can catch it. If there is no C++ code to catch it, the behavior is undefined.

The error handler `MPI::ERRORS_THROW_EXCEPTIONS` causes an `MPI::Exception` to be thrown for any MPI result code other than `MPI::SUCCESS`.

The C++ bindings for exceptions follow:

```
namespace MPI [  
  
    Exception::Exception(int error_code);  
    int Exception::Get_error_code() const;  
    int Exception::Get_error_class() const;  
    const char* Exception::Get_error_string() const;  
  
];
```

The public interface to `MPI::Exception` class is defined as follows:

```
namespace MPI [  
    class Exception [  
    public:  
  
        Exception(int error_code);  
  
        int Get_error_code() const;  
        int Get_error_class() const;  
        const char *Get_error_string() const;  
    ];  
];
```

The PE MPI implementation follows:

```
public:  
  
    Exception(int ec) : error_code(ec)  
    [  
        (void)MPI_Error_class(error_code, &error_class);  
        int resultlen;  
        (void)MPI_Error_string(error_code, error_string, &resultlen);  
    ]  
  
    virtual ~Exception(){ }  
  
    virtual int Get_error_code() const  
    [  

```

```

        return error_code;
    ]

    virtual int Get_error_class() const
    [
        return error_class;
    ]

    virtual const char* Get_error_string() const
    [
        return error_string;
    ]

protected:

    int error_code;
    char error_string[MPI_MAX_ERROR_STRING];
    int error_class;

};

```

---

## Chapter 7. Predefined MPI datatypes

This chapter lists the predefined MPI datatypes that you can use with MPI:

- “Special purpose datatypes”
- “Datatypes for C language bindings”
- “Datatypes for FORTRAN language bindings” on page 50
- “Datatypes for reduction functions (C reduction types)” on page 50
- “Datatypes for reduction functions (FORTRAN reduction types)” on page 51

---

### Special purpose datatypes

*Table 5. Special purpose datatypes*

Datatype	Description
MPI_BYTE	Untyped byte data
MPI_LB	Explicit lower bound marker
MPI_PACKED	Packed data (byte)
MPI_UB	Explicit upper bound marker

---

### Datatypes for C language bindings

*Table 6. Datatypes for C language bindings*

Datatype	Description
MPI_CHAR	8-bit character
MPI_DOUBLE	64-bit floating point
MPI_FLOAT	32-bit floating point
MPI_INT	32-bit integer
MPI_LONG	32-bit integer
MPI_LONG_DOUBLE	64-bit floating point
MPI_LONG_LONG	64-bit integer
MPI_LONG_LONG_INT	64-bit integer
MPI_SHORT	16-bit integer
MPI_SIGNED_CHAR	8-bit signed character
MPI_UNSIGNED	32-bit unsigned integer
MPI_UNSIGNED_CHAR	8-bit unsigned character
MPI_UNSIGNED_LONG	32-bit unsigned integer
MPI_UNSIGNED_LONG_LONG	64-bit unsigned integer
MPI_UNSIGNED_SHORT	16-bit unsigned integer
MPI_WCHAR	Wide (16-bit) unsigned character

---

## Datatypes for FORTRAN language bindings

Table 7. Datatypes for FORTRAN language bindings

Datatype	Description
MPI_CHARACTER	8-bit character
MPI_COMPLEX	32-bit floating point real, 32-bit floating point imaginary
MPI_COMPLEX8	32-bit floating point real, 32-bit floating point imaginary
MPI_COMPLEX16	64-bit floating point real, 64-bit floating point imaginary
MPI_COMPLEX32	128-bit floating point real, 128-bit floating point imaginary
MPI_DOUBLE_COMPLEX	64-bit floating point real, 64-bit floating point imaginary
MPI_DOUBLE_PRECISION	64-bit floating point
MPI_INTEGER	32-bit integer
MPI_INTEGER1	8-bit integer
MPI_INTEGER2	16-bit integer
MPI_INTEGER4	32-bit integer
MPI_INTEGER8	64-bit integer
MPI_LOGICAL	32-bit logical
MPI_LOGICAL1	8-bit logical
MPI_LOGICAL2	16-bit logical
MPI_LOGICAL4	32-bit logical
MPI_LOGICAL8	64-bit logical
MPI_REAL	32-bit floating point
MPI_REAL4	32-bit floating point
MPI_REAL8	64-bit floating point
MPI_REAL16	128-bit floating point

---

## Datatypes for reduction functions (C reduction types)

Table 8. Datatypes for reduction functions (C reduction types)

Datatype	Description
MPI_DOUBLE_INT	{MPI_DOUBLE, MPI_INT}
MPI_FLOAT_INT	{MPI_FLOAT, MPI_INT}
MPI_LONG_DOUBLE_INT	{MPI_LONG_DOUBLE, MPI_INT}
MPI_LONG_INT	{MPI_LONG, MPI_INT}
MPI_SHORT_INT	{MPI_SHORT, MPI_INT}
MPI_2INT	{MPI_INT, MPI_INT}

---

## Datatypes for reduction functions (FORTRAN reduction types)

Table 9. Datatypes for reduction functions (FORTRAN reduction types)

Datatype	Description
MPI_2COMPLEX	{MPI_COMPLEX, MPI_COMPLEX}
MPI_2DOUBLE_PRECISION	{MPI_DOUBLE_PRECISION, MPI_DOUBLE_PRECISION}
MPI_2INTEGER	{MPI_INTEGER, MPI_INTEGER}
MPI_2REAL	{MPI_REAL, MPI_REAL}



---

## Chapter 8. MPI reduction operations

The chapter describes predefined MPI reduction operations, including their datatypes, and provides C and FORTRAN examples.

---

### Predefined operations

Table 10 lists the predefined operations for use with MPI\_ALLREDUCE, MPI\_REDUCE, MPI\_REDUCE\_SCATTER and MPI\_SCAN. To invoke a predefined operation, place any of the following reductions in *op*.

Table 10. Predefined reduction operations

Operation	Description
MPI_BAND	bitwise AND
MPI BOR	bitwise OR
MPI_BXOR	bitwise XOR
MPI LAND	logical AND
MPI_LOR	logical OR
MPI_LXOR	logical XOR
MPI_MAX	maximum value
MPI_MAXLOC	maximum value and location
MPI_MIN	minimum value
MPI_MINLOC	minimum value and location
MPI_PROD	product
MPI_REPLACE	$f(a,b) = b$ (the current value in the target memory is replaced by the value supplied by the origin)
MPI_SUM	sum

### Datatype arguments of reduction operations

Table 11 lists the basic datatype arguments of the reduction operations.

Table 11. Valid datatype arguments

Type	Arguments
Byte	MPI_BYTE
C integer	MPI_INT MPI_LONG MPI_LONG_LONG_INT MPI_SHORT MPI_UNSIGNED MPI_UNSIGNED_LONG MPI_UNSIGNED_LONG_LONG MPI_UNSIGNED_SHORT

Table 11. Valid datatype arguments (continued)

Type	Arguments
C pair	MPI_DOUBLE_INT MPI_FLOAT_INT MPI_LONG_INT MPI_LONG_DOUBLE_INT MPI_SHORT_INT MPI_2INT
Complex	MPI_COMPLEX
Floating point	MPI_DOUBLE MPI_DOUBLE_PRECISION MPI_FLOAT MPI_LONG_DOUBLE MPI_REAL
FORTRAN integer	MPI_INTEGER MPI_INTEGER8
FORTRAN pair	MPI_2DOUBLE_PRECISION MPI_2INTEGER MPI_2REAL
Logical	MPI_LOGICAL

## Valid datatypes for the op option

Table 12 lists the valid datatypes for each **op** option.

Table 12. Valid datatypes for the op option

Type	Datatypes
Byte	MPI_BAND MPI BOR MPI_BXOR MPI_REPLACE
C integer	MPI_BAND MPI BOR MPI_BXOR MPI LAND MPI_LOR MPI_LXOR MPI_MAX MPI_MIN MPI_PROD MPI_REPLACE MPI_SUM
C pair	MPI_MAXLOC MPI_MINLOC MPI_REPLACE
Complex	MPI_PROD MPI_REPLACE MPI_SUM
Floating point	MPI_MAX MPI_MIN MPI_PROD MPI_REPLACE MPI_SUM

Table 12. Valid datatypes for the op option (continued)

Type	Datatypes
FORTTRAN integer	MPI_BAND MPI_BOR MPI_BXOR MPI_MAX MPI_MIN MPI_PROD MPI_REPLACE MPI_SUM
FORTTRAN pair	MPI_MAXLOC MPI_MINLOC MPI_REPLACE
Logical	MPI_LAND MPI_LOR MPI_LXOR MPI_REPLACE

## Examples

Examples of user-defined reduction functions for integer vector addition follow.

### C example

```
void int_sum (int *in, int *inout,
             int *len, MPI_Datatype *type);

{
    int i
    for (i=0; i<*len; i++) {
        inout[i] += in[i];
    }
}
```

### FORTTRAN example

```
SUBROUTINE INT_SUM(IN,INOUT,LEN,TYPE)

    INTEGER IN(*),INOUT(*),LEN,TYPE,I

    DO I = 1,LEN
        INOUT(I) = IN(I) + INOUT(I)
    ENDDO
END
```

User-supplied reduction operations have four arguments:

- The first argument, **in**, is an array or scalar variable. The length, in elements, is specified by the third argument, **len**.  
This argument is an input array to be reduced.
- The second argument, **inout**, is an array or scalar variable. The length, in elements, is specified by the third argument, **len**.  
This argument is an input array to be reduced and the result of the reduction will be placed here.
- The third argument, **len** is the number of elements in **in** and **inout** to be reduced.
- The fourth argument **type** is the datatype of the elements to be reduced.

Users may code their own reduction operations, with the restriction that the operations must be associative. Also, C programmers should note that the values of **len** and **type** will be passed as pointers. No communication calls are allowed in user-defined reduction operations. See “Limitations in setting the thread stack size” on page 37 for thread stack size considerations when using the MPI threads library.

---

## Chapter 9. C++ MPI constants

This chapter lists C++ MPI constants, including the following:

- “Error classes”
- “Maximum sizes” on page 58
- “Environment inquiry keys” on page 58
- “Predefined attribute keys” on page 59
- “Results of communicator and group comparisons” on page 59
- “Topologies” on page 59
- “File operation constants” on page 59
- “MPI-IO constants” on page 59
- “One-sided constants” on page 60
- “Combiner constants used for datatype decoding functions” on page 60
- “Assorted constants” on page 60
- “Collective constants” on page 60
- “Error handling specifiers” on page 60
- “Special datatypes for construction of derived datatypes” on page 61
- “Elementary datatypes (C and C++)” on page 61
- “Elementary datatypes (FORTRAN)” on page 61
- “Datatypes for reduction functions (C and C++)” on page 61
- “Datatypes for reduction functions (FORTRAN)” on page 61
- “Optional datatypes” on page 62
- “Collective operations” on page 62
- “Null handles” on page 62
- “Empty group” on page 62
- “Threads constants” on page 63
- “FORTRAN 90 datatype matching constants” on page 63

---

### Error classes

MPI::SUCCESS  
MPI::ERR\_BUFFER  
MPI::ERR\_COUNT  
MPI::ERR\_TYPE  
MPI::ERR\_TAG  
MPI::ERR\_COMM  
MPI::ERR\_RANK  
MPI::ERR\_REQUEST  
MPI::ERR\_ROOT  
MPI::ERR\_GROUP  
MPI::ERR\_OP  
MPI::ERR\_TOPOLOGY  
MPI::ERR\_DIMS  
MPI::ERR\_ARG  
MPI::ERR\_UNKNOWN  
MPI::ERR\_TRUNCATE

MPI::ERR\_OTHER  
MPI::ERR\_INTERN  
MPI::ERR\_IN\_STATUS  
MPI::ERR\_PENDING  
MPI::ERR\_INFO\_KEY  
MPI::ERR\_INFO\_VALUE  
MPI::ERR\_INFO\_NOKEY  
MPI::ERR\_INFO  
MPI::ERR\_FILE  
MPI::ERR\_NOT\_SAME  
MPI::ERR\_AMODE  
MPI::ERR\_UNSUPPORTED\_DATAREP  
MPI::ERR\_UNSUPPORTED\_OPERATION  
MPI::ERR\_NO\_SUCH\_FILE  
MPI::ERR\_FILE\_EXISTS  
MPI::ERR\_BAD\_FILE  
MPI::ERR\_ACCESS  
MPI::ERR\_NO\_SPACE  
MPI::ERR\_QUOTA  
MPI::ERR\_READ\_ONLY  
MPI::ERR\_FILE\_IN\_USE  
MPI::ERR\_DUP\_DATAREP  
MPI::ERR\_CONVERSION  
MPI::ERR\_IO  
MPI::ERR\_WIN  
MPI::ERR\_BASE  
MPI::ERR\_SIZE  
MPI::ERR\_DISP  
MPI::ERR\_LOCKTYPE  
MPI::ERR\_ASSERT  
MPI::ERR\_RMA\_CONFLICT  
MPI::ERR\_RMA\_SYNC  
MPI::ERR\_NO\_MEM  
MPI::ERR\_LASTCODE

---

## Maximum sizes

MPI::MAX\_ERROR\_STRING  
MPI::MAX\_PROCESSOR\_NAME  
MPI::MAX\_FILE\_NAME  
MPI::MAX\_DATAREP\_STRING  
MPI::MAX\_INFO\_KEY  
MPI::MAX\_INFO\_VAL  
MPI::MAX\_OBJECT\_NAME

---

## Environment inquiry keys

MPI::TAG\_UB  
MPI::IO  
MPI::HOST  
MPI::WTIME\_IS\_GLOBAL

---

## Predefined attribute keys

MPI::LASTUSEDPCODE  
MPI::WIN\_BASE  
MPI::WIN\_SIZE  
MPI::WIN\_DISP\_UNIT

---

## Results of communicator and group comparisons

MPI::IDENT  
MPI::CONGRUENT  
MPI::SIMILAR  
MPI::UNEQUAL

---

## Topologies

MPI::GRAPH  
MPI::CART

---

## File operation constants

MPI::SEEK\_SET  
MPI::SEEK\_CUR  
MPI::SEEK\_END  
MPI::DISTRIBUTE\_NONE  
MPI::DISTRIBUTE\_BLOCK  
MPI::DISTRIBUTE\_CYCLIC  
MPI::DISTRIBUTE\_DFLT\_DARG  
MPI::ORDER\_C  
MPI::ORDER\_FORTRAN  
MPI::DISPLACEMENT\_CURRENT

---

## MPI-IO constants

MPI::MODE\_RDONLY  
MPI::MODE\_WRONLY  
MPI::MODE\_RDWR  
MPI::MODE\_CREATE  
MPI::MODE\_APPEND  
MPI::MODE\_EXCL  
MPI::MODE\_DELETE\_ON\_CLOSE  
MPI::MODE\_UNIQUE\_OPEN  
MPI::MODE\_SEQUENTIAL  
MPI::MODE\_NOCHECK  
MPI::MODE\_NOSTORE  
MPI::MODE\_NOPUT  
MPI::MODE\_NOPRECEDE  
MPI::MODE\_NOSUCCEED

---

## One-sided constants

MPI::LOCK\_EXCLUSIVE  
MPI::LOCK\_SHARED

---

## Combiner constants used for datatype decoding functions

MPI::COMBINER\_NAMED  
MPI::COMBINER\_DUP  
MPI::COMBINER\_CONTIGUOUS  
MPI::COMBINER\_VECTOR  
MPI::COMBINER\_HVECTOR\_INTEGER  
MPI::COMBINER\_HVECTOR  
MPI::COMBINER\_INDEXED  
MPI::COMBINER\_HINDEXED\_INTEGER  
MPI::COMBINER\_HINDEXED  
MPI::COMBINER\_INDEXED\_BLOCK  
MPI::COMBINER\_STRUCT\_INTEGER  
MPI::COMBINER\_STRUCT  
MPI::COMBINER\_SUBARRAY  
MPI::COMBINER\_DARRAY  
MPI::COMBINER\_F90\_REAL  
MPI::COMBINER\_F90\_COMPLEX  
MPI::COMBINER\_F90\_INTEGER  
MPI::COMBINER\_RESIZED

---

## Assorted constants

MPI::BSEND\_OVERHEAD  
MPI::PROC\_NULL  
MPI::ANY\_SOURCE  
MPI::ANY\_TAG  
MPI::UNDEFINED  
MPI::KEYVAL\_INVALID  
MPI::BOTTOM

---

## Collective constants

MPI::ROOT  
MPI::IN\_PLACE

---

## Error handling specifiers

MPI::ERRORS\_ARE\_FATAL  
MPI::ERRORS\_RETURN  
MPI::ERRORS\_THROW\_EXCEPTIONS  
(see “Predefined error handler for C++” on page 47)

---

## Special datatypes for construction of derived datatypes

MPI::UB  
MPI::LB  
MPI::BYTE  
MPI::PACKED

---

## Elementary datatypes (C and C++)

MPI::CHAR  
MPI::UNSIGNED\_CHAR  
MPI::SIGNED\_CHAR  
MPI::SHORT  
MPI::INT  
MPI::LONG  
MPI::UNSIGNED\_SHORT  
MPI::UNSIGNED  
MPI::UNSIGNED\_LONG  
MPI::FLOAT  
MPI::DOUBLE  
MPI::LONG\_DOUBLE  
MPI::LONG\_LONG  
MPI::UNSIGNED\_LONG\_LONG  
MPI::WCHAR

---

## Elementary datatypes (FORTRAN)

MPI::INTEGER  
MPI::REAL  
MPI::DOUBLE\_PRECISION  
MPI::F\_COMPLEX  
MPI::LOGICAL  
MPI::CHARACTER

---

## Datatypes for reduction functions (C and C++)

MPI::FLOAT\_INT  
MPI::DOUBLE\_INT  
MPI::LONG\_INT  
MPI::TWOINT  
MPI::SHORT\_INT  
MPI::LONG\_DOUBLE\_INT

---

## Datatypes for reduction functions (FORTRAN)

MPI::TWOREAL  
MPI::TWODOUBLE\_PRECISION  
MPI::TWOINTEGER  
MPI::TWOOF\_COMPLEX

---

## Optional datatypes

MPI::INTEGER1  
MPI::INTEGER2  
MPI::INTEGER4  
MPI::INTEGER8  
MPI::REAL4  
MPI::REAL8  
MPI::REAL16  
MPI::LOGICAL1  
MPI::LOGICAL2  
MPI::LOGICAL4  
MPI::LOGICAL8  
MPI::F\_DOUBLE\_COMPLEX  
MPI::F\_COMPLEX8  
MPI::F\_COMPLEX16  
MPI::F\_COMPLEX32

---

## Collective operations

MPI::MAX  
MPI::MIN  
MPI::SUM  
MPI::PROD  
MPI::MAXLOC  
MPI::MINLOC  
MPI::BAND  
MPI::BOR  
MPI::BXOR  
MPI::LAND  
MPI::LOR  
MPI::LXOR  
MPI::REPLACE

---

## Null handles

MPI::GROUP\_NULL  
MPI::COMM\_NULL  
MPI::DATATYPE\_NULL  
MPI::REQUEST\_NULL  
MPI::OP\_NULL  
MPI::ERRHANDLER\_NULL  
MPI::INFO\_NULL  
MPI::WIN\_NULL

---

## Empty group

MPI::GROUP\_EMPTY

---

## Threads constants

```
MPI::THREAD_SINGLE  
MPI::THREAD_FUNNELED  
MPI::THREAD_SERIALIZED  
MPI::THREAD_MULTIPLE
```

---

## FORTRAN 90 datatype matching constants

```
MPI::TYPECLASS_REAL  
MPI::TYPECLASS_INTEGER  
MPI::TYPECLASS_COMPLEX
```



---

## Chapter 10. MPI size limits

This chapter gives limitations for MPI elements and parallel job tasks, including:

- “System limits.”
- “Maximum number of tasks and tasks per node” on page 67.

---

### System limits

The following list includes system limits on the size of various MPI elements and the relevant environment variable or tunable parameter. The MPI standard identifies several values that have limits in any MPI implementation. For these values, the standard indicates a named constant to express the limit. See **mpi.h** for these constants and their values. The limits described below are specific to PE and are not part of standard MPI.

- Number of tasks: **MP\_PROCS**
- Maximum number of tasks: 8192
- Maximum buffer size for any MPI communication (for 32-bit applications only): 2 GB
- Default early arrival buffer size: (**MP\_BUFFER\_MEM**)
  - When using Internet Protocol (IP): 2 800 000 bytes
  - When using User Space: 64 MB
- Minimum pre-allocated early arrival buffer size: (50 \* eager\_limit) number of bytes
- Maximum pre-allocated early arrival buffer size: 256 MB
- Minimum message envelope buffer size: 1 MB
- Default eager limit (**MP\_EAGER\_LIMIT**): See Table 13 on page 66. Note that the default values shown in Table 13 on page 66 are initial estimates that are used by the MPI library. Depending on the value of **MP\_BUFFER\_MEM** and the job type, these values will be adjusted to guarantee a safe eager send protocol.
- Maximum eager limit: 256 KB
- MPI uses the **MP\_BUFFER\_MEM** and the **MP\_EAGER\_LIMIT** values that are selected for a job to determine how many complete messages, each with a size that is equal to or less than the eager\_limit, can be sent eagerly from every task of the job to a single task, without causing the single target to run out of buffer space. This is done by allocating to each sending task a number of message credits for each target. The sending task will consume one message credit for each eager send to a particular target. It will get that credit back after the message has been matched at that target.

The sending task can continue to send eager messages to a particular target as long as it still has message credits for that target. The following equation is used to calculate the number of credits to be allocated:

$$\text{MP\_BUFFER\_MEM} / (\text{MP\_PROCS} * \text{MAX}(\text{MP\_EAGER\_LIMIT}, 64))$$

MPI uses this equation to ensure that there are at least two credits for each target. If needed, MPI reduces the initially selected value of **MP\_EAGER\_LIMIT**, or increases the initially selected value of **MP\_BUFFER\_MEM**, in order to achieve this minimum threshold of two credits for each target.

If the user has specified an initial value for **MP\_BUFFER\_MEM** or **MP\_EAGER\_LIMIT**, and MPI has changed either one or both of these values,

an informational message is issued. If the user has specified **MP\_BUFFER\_MEM** using the two values format, then the maximum value specified by the second parameter will be used in the equation above. See *IBM Parallel Environment 4.2: Operation and Use, Volume 1* for more information about specifying values for **MP\_BUFFER\_MEM**.

If the user allows both **MP\_BUFFER\_MEM** and **MP\_EAGER\_LIMIT** to default, then the initial value that was selected for **MP\_BUFFER\_MEM** will be 64 MB for a User Space job and 2.8 MB for an IP job. MPI estimates the initial value for **MP\_EAGER\_LIMIT** based on the job size, as shown in Table 13. MPI then does the calculation again to ensure that there will be at least two credits for each target.

For example, with the defaults of both **MP\_BUFFER\_MEM** and **MP\_EAGER\_LIMIT** then, according to the equation, each sending task of an 8192 task User Space job will have a minimum of 8 credits for each target. Each sending task of a 4096 task User Space job will have a minimum of 16 complete credits for each target. However, for an IP job, because of the smaller **MP\_BUFFER\_MEM** value, only jobs with less than 11 tasks per job will have more than two credits allocated for each target. Most IP jobs can have only two credits for each target, and even this can be accomplished only by greatly reducing the value of the **MP\_EAGER\_LIMIT**. For example, each sending task of an 8192 tasks IP job can have only two 128 byte credits for each target, including the task itself. For an IP job you should consider increasing **MP\_BUFFER\_MEM** above the 2.8 MB default, unless memory is very limited.

Any time a message that is small enough to be eligible for eager send cannot be guaranteed destination buffer space, the message is handled by rendezvous protocol. Destination buffer space unavailability cannot cause a safe MPI program to fail, but could cause hangs in unsafe MPI programs. An *unsafe* program is one that assumes MPI can guarantee system buffering of sent data until the receive is posted. The MPI standard warns that unsafe programs, though they may work in some cases, are not valid MPI. We suggest every application be checked for safety by running it just once with **MP\_EAGER\_LIMIT** set to 0, which will cause an unsafe application to hang. Because eager limit, along with task count, affects the minimum buffer memory requirement, it is possible to produce an unworkable combination when both **MP\_EAGER\_LIMIT** and **MP\_BUFFER\_MEM** are explicitly set. MPI will override unworkable combinations. If either the **MP\_EAGER\_LIMIT** or the **MP\_BUFFER\_MEM** value is changed by MPI, an informational message is issued.

Table 13. MPI eager limits

Number of tasks	Default limit (MP_EAGER_LIMIT)
1 to 256	32768
257 to 512	16384
513 to 1024	8192
1025 to 2048	4096
2049 to 4096	2048
4097 to 8192	1024

- Maximum aggregate unsent data, per task: no specific limit
- Maximum number of communicators, file handles, and windows: approximately 2000
- Maximum number of distinct tags: all nonnegative integers less than  $2^{32}-1$

## Maximum number of tasks and tasks per node

The following table lists the limits on the total number of tasks in a parallel job, and the maximum number of tasks on a node (operating system image). If two limits are listed, the most restrictive limit applies.

Table 14. Task limits for parallel jobs

Protocol Library	Switch/Adapter	Total Task Limit	Task per Node Limit
IP	any	8192	large
User Space	SP Switch2/PCI-X	8192	32
User Space	SP Switch2/PCI	8192	32
User Space	pSeries HPS with one adapter	8192	64
User Space	pSeries HPS with two adapters per network	8192	128
User Space	SP Switch2/SW2	8192	16

For a system with a pSeries HPS switch and adapter, the *Task per Node Limit* is 64 tasks per adapter per network. For a system with two adapters per network, the task per node limit is 128, or  $64 * 2$ . This enables the running of a 128 task per node MPI job over User Space. This may be useful on 64 CPU nodes with the Simultaneous Multi-Threading (SMT) technology available on POWER5 and AIX 5.3 enabled. The LoadLeveler configuration also helps determine how many tasks can be run on a node. To run 128 tasks per node, LoadLeveler must be configured with 128 starters per node. In theory, you can configure more than two adapters per network and run more than 128 tasks per node. However, this means running more than one task per CPU, and results in reduced performance. Also, the lower layer of the protocol stack has a 128 tasks per node limit for enabling shared memory. The protocol stack does not use shared memory when there are more than 128 tasks per node.

For running an MPI job over IP, the task per node limit is not affected by the number of adapters; the task per node limit is determined only by the number of LoadLeveler starters configured per node. The 128 task per node limit for enabling shared memory usage also applies to MPI/IP jobs.

Although the PCI adapters support the stated limits for tasks per node, maximum aggregate bandwidth through the adapter is achieved with a smaller task per node count, if all tasks are simultaneously involved in message passing. Thus, if individual MPI tasks can do SMP parallel computations on multiple CPUs (using OpenMP or threads), performance may be better than if all MPI tasks compete for adapter resources.

The user may also want to consider using MPI IP. On SP Switch2 PCI systems with many MPI tasks sharing adapters, MPI IP may perform better than MPI User Space.



---

## Chapter 11. POE environment variables and command-line flags

This section contains tables which summarize the environment variables and command-line flags discussed throughout this book. You can set these variables and flags to influence the execution of parallel programs, and the operation of certain tools. The command-line flags temporarily override their associated environment variable. The tables divide the environment variables and flags by function:

- Table 15 on page 70 summarizes the environment variables and flags for controlling the Partition Manager. These environment variables and flags enable you to specify such things as an input or output host list file, and the method of node allocation.
- Table 16 on page 73 summarizes the environment variables and flags for Job Specifications. These environment variables and flags determine whether or not the Partition Manager should maintain the partition for multiple job steps, whether commands should be read from a file or STDIN, and how the partition should be loaded.
- Table 17 on page 74 summarizes the environment variables and flags for determining how I/O from the parallel tasks should be handled. These environment variables and flags set the input and output modes, and determine whether or not output is labeled by task id.
- Table 18 on page 76 summarizes the environment variables and flags for collecting diagnostic information. These environment variables and flags enable you to generate diagnostic information that may be required by the IBM Support Center in resolving PE-related problems.
- Table 19 on page 76 summarizes the environment variables and flags for the Message Passing Interface. These environment variables and flags allow you to change message and memory sizes, as well as other message passing information.
- Table 20 on page 83 summarizes the variables and flags for core file generation.
- Table 21 on page 83 summarizes some miscellaneous environment variables and flags. These environment variables and flags provide control for the Program Marker Array, enable additional error checking, and let you set a dispatch priority class for execution.

You can use the POE command-line flags on the **poe** and **pdbx** commands. You can also use the following flags on program names when individually loading nodes from STDIN or a POE commands file.

- **-infolevel** or **-ilevel**
- **-euidevelop**

In the tables that follow, a check mark (✓) denotes those flags you can use when individually loading nodes.

Table 15. POE environment variables and command-line flags for partition manager control

The Environment Variable/Command-Line Flag(s):	Set:	Possible Values:	Default:
MP_ADAPTER_USE -adapter_use	How the node's adapter should be used. The User Space communication subsystem library does not require dedicated use of the high performance switch on the node. Adapter use will be defaulted, but shared usage may be specified.	One of the following strings:  <i>dedicated</i> Only a single program task can use the adapter.  <i>shared</i> A number of tasks on the node can use the adapter.	Dedicated for User Space jobs, shared for IP jobs.
MP_CPU_USE -cpu_use	How the node's CPU should be used. The User Space communication subsystem library does not require unique CPU use on the node. CPU use will be defaulted, but multiple use may be specified.  For example, either one job per node gets all CPUs, or more than one job can go on a node.	One of the following strings:  <i>unique</i> Only your program's tasks can use the CPU.  <i>multiple</i> Your program may share the node with other users.	Unique for User Space jobs, multiple for IP jobs.
MP_EUIDEVICE -euidevice	The adapter set to use for message passing – either Ethernet, FDDI, token-ring, the IBM RS/6000 SP's high performance switch adapter, the SP switch 2, or the pSeries High Performance Switch.	One of the following strings:  <i>en0</i> Ethernet <i>fi0</i> FDDI <i>tr0</i> token-ring <i>css0</i> high performance switch <i>csss</i> SP switch 2 <i>sn_all</i> <i>sn_single</i> <i>ml0</i>	The adapter set used as the external network address.
MP_EUILIB -eulib	The communication subsystem implementation to use for communication – either the IP communication subsystem or the User Space communication subsystem.	One of the following strings:  <i>ip</i> The IP communication subsystem.  <i>us</i> The User Space communication subsystem. <b>Note:</b> This specification is case-sensitive.	ip
MP_EUILIBPATH -eulibpath	The path to the message passing and communication subsystem libraries. This only needs to be set if the libraries are moved, or an alternate set is being used.	Any path specifier.	/usr/lpp/ppe.poe/lib

Table 15. POE environment variables and command-line flags for partition manager control (continued)

The Environment Variable/Command-Line Flag(s):	Set:	Possible Values:	Default:
MP_HOSTFILE -hostfile -hfile	The name of a host list file for node allocation.	Any file specifier or the word NULL.	<i>host.list</i> in the current directory.
MP_INSTANCES -instances	The number of instances of User Space windows or IP addresses to be assigned. This value is expressed as an integer, or the string <b>max</b> . If the values specified exceeds the maximum allowed number of instances, as determined by LoadLeveler, that number is substituted.	A positive integer, or the string <b>max</b> .	1
MP_PROCS -procs	The number of program tasks.	Any number from 1 to the maximum supported configuration.	1
MP_PULSE -pulse	The interval (in seconds) at which POE checks the remote nodes to ensure that they are actively communicating with the home node. <b>Note:</b> <b>Pulse</b> is ignored for <b>pdbx</b> .	An integer greater than or equal to 0.	600
MP_RESD -resd	Whether or not the Partition Manager should connect to LoadLeveler to allocate nodes. <b>Note:</b> When running POE from a workstation that is external to the LoadLeveler cluster, the <i>LoadL.so</i> fileset must be installed on the external node (see <i>Using and Administering LoadLeveler</i> and <i>IBM Parallel Environment for AIX: Installation</i> for more information).	<i>yes no</i>	Context dependent
MP_RETRY -retry	The period of time (in seconds) between processor node allocation retries by POE if there are not enough processor nodes immediately available to run a program. This is valid only if you are using LoadLeveler. If the character string <b>wait</b> is specified instead of a number, no retries are attempted by POE, and the job remains enqueued in LoadLeveler until LoadLeveler either schedules the job or cancels it.	An integer greater than or equal to 0, or the case-insensitive value <b>wait</b> .	0 (no retry)
MP_RETRYCOUNT -retrycount	The number of times (at the interval set by <b>MP_RETRY</b> ) that the partition manager should attempt to allocate processor nodes. This value is ignored if <b>MP_RETRY</b> is set to the character string <b>wait</b> .	An integer greater than or equal to 0.	0

Table 15. POE environment variables and command-line flags for partition manager control (continued)

The Environment Variable/Command-Line Flag(s):	Set:	Possible Values:	Default:
MP_MSG_API -msg_api	<p>To indicate to POE which message passing API is being used by the application code.</p> <p><b>MPI</b> Indicates that the application makes only MPI calls.</p> <p><b>LAPI</b> Indicates that the application makes only LAPI calls.</p> <p><b>MPI_LAPI</b> Indicates that calls to both message passing APIs are used in the application, and the same set of communication resources (windows, IP addresses) is to be shared between them.</p> <p><b>MPI,LAPI</b> Indicates that calls to both message passing APIs are used in the application, with dedicated resources assigned to each of them.</p> <p><b>LAPI,MPI</b> Has a meaning identical to <b>MPI,LAPI</b>.</p>	<p><b>MPI</b> <b>LAPI</b> <b>MPI_LAPI</b> <b>MPI,LAPI</b> <b>LAPI,MPI</b></p>	<b>MPI</b>
MP_RMPOOL -rmpool	The name or number of the pool that should be used for nonspecific node allocation. This environment variable/command-line flag only applies to LoadLeveler.	An identifying pool name or number.	None
MP_NODES -nodes	To specify the number of processor nodes on which to run the parallel tasks. It may be used alone or in conjunction with <b>MP_TASKS_PER_NODE</b> and/or <b>MP_PROCS</b> .	Any number from 1 to the maximum supported configuration.	None
MP_TASKS_PER_NODE -tasks_per_node	To specify the number of tasks to be run on each of the physical nodes. It may be used in conjunction with <b>MP_NODES</b> and/or <b>MP_PROCS</b> , but may not be used alone.	Any number from 1 to the maximum supported configuration.	None
MP_SAVEHOSTFILE -savehostfile	The name of an output host list file to be generated by the Partition Manager.	Any relative or full path name.	None
MP_REMOTEDIR (no associated command line flag)	The name of a script which echoes the name of the current directory to be used on the remote nodes.	Any file specifier.	None
MP_TIMEOUT (no associated command line flag)	The length of time that POE waits before abandoning an attempt to connect to the remote nodes.	Any number greater than 0. If set to 0 or a negative number, the value is ignored.	150 seconds

Table 15. POE environment variables and command-line flags for partition manager control (continued)

The Environment Variable/Command-Line Flag(s):	Set:	Possible Values:	Default:
MP_CKPTFILE  (no associated command line flag)	The base name of the checkpoint file.	Any file specifier.	
MP_CKPTDIR  (no associated command line flag)	The directory where the checkpoint file will reside.	Any path specifier.	Directory from which POE is run.

Table 16. POE environment variables/command-line flags for job specification

The Environment Variable/Command-Line Flag(s):	Set:	Possible Values:	Default:
MP_CMDFILE  -cmdfile	The name of a POE commands file used to load the nodes of your partition. If set, POE will read the commands file rather than STDIN.	Any file specifier.	None
MP_LLFILE  -llfile	The name of a LoadLeveler job command file for node allocation. If you are performing specific node allocation, you can use a LoadLeveler job command file in conjunction with a host list file. If you do, the specific nodes listed in the host list file will be requested from LoadLeveler.	Any path specifier.	None
MP_NEWJOB  -newjob	Whether or not the Partition Manager maintains your partition for multiple job steps.	<i>yes no</i>	<i>no</i>
MP_PGMMODEL  -pgmmodel	The programming model you are using.	<i>spmd mpmc</i>	<i>spmd</i>
MP_SAVE_LLFILE  -save_llfile	When using LoadLeveler for node allocation, the name of the output LoadLeveler job command file to be generated by the Partition Manager. The output LoadLeveler job command file will show the LoadLeveler settings that result from the POE environment variables and/or command-line options for the current invocation of POE. If you use the <b>MP_SAVE_LLFILE</b> environment variable for a batch job, or when the <b>MP_LLFILE</b> environment variable is set (indicating that a LoadLeveler job command file should participate in node allocation), POE will show a warning and will not save the output job command file.	Any relative or full path name.	None

Table 16. POE environment variables/command-line flags for job specification (continued)

The Environment Variable/Command-Line Flag(s):	Set:	Possible Values:	Default:
MP_TASK_AFFINITY -task_affinity	<p>This causes the PMD to attach each task of a parallel job to one of the system resource sets (rsets) at the MCM level, thus constraining the task (and all its threads) to run within that MCM. If the task has an inherited resource set, the attach honors the constraints of the inherited resource set.</p> <p>It is recommended that the user also set the AIX environment variable <b>MEMORY_AFFINITY</b> to <b>MCM</b>.</p>	<p><i>SNI</i> Specifies that the PMD select the MCM to which the first adapter window is attached.</p> <p><i>MCM</i> Specifies that the PMD assigns tasks on a round-robin basis to the MCMs in the inherited resource set. If WLM is not being used, this is most useful when a node is being used for only one job.</p> <p><i>-1</i> Specifies that no affinity request is to be made.</p> <p><i>mcm_list</i> Specifies a set of system level (LPAR) logical MCMs that can be attached to. Tasks of this job will be assigned round-robin to this set, within the constraint of an inherited rset, if any. Any MCMs outside the constraint set will be attempted, but fail.</p>	None

Table 17. POE environment variables/command-line flags for I/O control

The Environment Variable/Command-Line Flag(s):	Set:	Possible Values:	Default:
MP_LABELIO -labelio	Whether or not output from the parallel tasks is labeled by task id.	<i>yes no</i>	<i>no</i> (yes for <b>pdbx</b> )

Table 17. POE environment variables/command-line flags for I/O control (continued)

The Environment Variable/Command-Line Flag(s):	Set:	Possible Values:	Default:
MP_STDINMODE -stdinmode	The input mode. This determines how input is managed for the parallel tasks.	<p><i>all</i> All tasks receive the same input data from STDIN.</p> <p><i>none</i> No tasks receive input data from STDIN; STDIN will be used by the home node only.</p> <p><b>a task id</b> STDIN is only sent to the task identified.</p>	<i>all</i>
MP_HOLD_STDIN (no associated command line flag)	Whether or not sending of STDIN from the home node to the remote nodes is deferred until the message passing partition has been established.	<i>yes no</i>	<i>no</i>
MP_STDOUTMODE -stdoutmode	The output mode. This determines how STDOUT is handled by the parallel tasks.	<p>One of the following:</p> <p><i>unordered</i> All tasks write output data to STDOUT asynchronously.</p> <p><i>ordered</i> Output data from each parallel task is written to its own buffer. Later, all buffers are flushed, in task order, to STDOUT.</p> <p><b>a task id</b> Only the task indicated writes output data to STDOUT.</p>	<i>unordered</i>

Table 18. POE environment variables/command-line flags for diagnostic information

The Environment Variable/Command-Line Flag(s):	Set:	Possible Values:	Default:
MP_INFOLEVEL -infolevel ✓ -ilevel ✓	The level of message reporting.	One of the following integers:  0 Error 1 Warning and error 2 Informational, warning, and error 3 Informational, warning, and error. Also reports high-level diagnostic messages for use by the IBM Support Center.  4, 5, 6 Informational, warning, and error. Also reports high- and low-level diagnostic messages for use by the IBM Support Center.	1
MP_PMDLOG -pmdlog	Whether or not diagnostic messages should be logged to a file in <i>/tmp</i> on each of the remote nodes. Typically, this environment variable/command-line flag is only used under the direction of the IBM Support Center in resolving a PE-related problem.	<i>yes no</i>	<i>no</i>
MP_DEBUG_INITIAL_STOP  (no associated command-line flag)	The initial breakpoint in the application where <b>pdbx</b> will get control.	One of the following: <i>"filename":line_number</i> <i>function_name</i>	The first executable source line in the main routine.
MP_DEBUG_NOTIMEOUT -debug_notimeout	A debugging aid that allows programmers to attach to one or more of their tasks without the concern that some other task may reach a timeout.	<b>Any non-null string</b> will activate this flag.	<b>no</b>

Table 19. POE environment variables and command-line flags for Message Passing Interface (MPI)

Environment Variable Command-Line Flag	Set:	Possible Values:	Default:
MP_ACK_THRESH -ack_thresh	Allows the user to control the packet acknowledgement threshold. Specify a positive integer.	A positive integer limited to 31	30

Table 19. POE environment variables and command-line flags for Message Passing Interface (MPI) (continued)

Environment Variable Command-Line Flag	Set:	Possible Values:	Default:
MP_BUFFER_MEM -buffer_mem	See "MP_BUFFER_MEM details" on page 82.		64 MB (User Space) 2800000 bytes (IP)
MP_CC_SCRATCH_BUF -cc_scratch_buf	Use the fastest collective communication algorithm even if that algorithm requires allocation of more scratch buffer space.	yes no	yes
MP_CLOCK_SOURCE -clock_source	To use the high performance switch clock as a time source. See "Using a switch clock as a time source" on page 34.	AIX SWITCH	None. See Table 3 on page 35.
MP_CSS_INTERRUPT -css_interrupt	To specify whether or not arriving packets generate interrupts. Using this environment variable may provide better performance for certain applications. Setting this variable explicitly will suppress the MPI-directed switching of interrupt mode, leaving the user in control for the rest of the run. For more information, see <b>MPI_FILE_OPEN</b> in <i>IBM Parallel Environment for AIX: MPI Subroutine Reference</i> .	yes no	no

Table 19. POE environment variables and command-line flags for Message Passing Interface (MPI) (continued)

Environment Variable Command-Line Flag	Set:	Possible Values:	Default:														
MP_EAGER_LIMIT -eager_limit	<p>To change the threshold value for message size, above which rendezvous protocol is used.</p> <p>To ensure that at least 32 messages can be outstanding between any two tasks, <b>MP_EAGER_LIMIT</b> will be adjusted based on the number of tasks according to the following table, when the user has specified neither <b>MP_BUFFER_MEM</b> nor <b>MP_EAGER_LIMIT</b>:</p> <table><tr><th>Number of Tasks</th><th>MP_EAGER_LIMIT</th></tr><tr><td>1 to 256</td><td>32768</td></tr><tr><td>257 to 512</td><td>16384</td></tr><tr><td>513 to 1024</td><td>8192</td></tr><tr><td>1025 to 2048</td><td>4096</td></tr><tr><td>2049 to 4096</td><td>2048</td></tr><tr><td>4097 to 8192</td><td>1024</td></tr></table> <p>The maximum value for <b>MP_EAGER_LIMIT</b> is 256 KB (262144 bytes). Any value that is less than 64 bytes but greater than zero bytes is automatically increased to 64 bytes. A value of zero bytes is valid, and indicates that eager send mode is not to be used for the job.</p>	Number of Tasks	MP_EAGER_LIMIT	1 to 256	32768	257 to 512	16384	513 to 1024	8192	1025 to 2048	4096	2049 to 4096	2048	4097 to 8192	1024	<p>nnnnn</p> <p>nnK (where: K = 1024 bytes)</p>	4096
Number of Tasks	MP_EAGER_LIMIT																
1 to 256	32768																
257 to 512	16384																
513 to 1024	8192																
1025 to 2048	4096																
2049 to 4096	2048																
4097 to 8192	1024																
MP_HINTS_FILTERED -hints_filtered	To specify whether or not MPI <b>info</b> objects reject hints ( <i>key</i> and <i>value</i> pairs) that are not meaningful to the MPI implementation.	<b>yes</b> <b>no</b>	<b>yes</b>														
MP_IONODEFILE -ionodefile	To specify the name of a parallel I/O node file — a text file that lists the nodes that should be handling parallel I/O. Setting this variable enables you to limit the number of nodes that participate in parallel I/O and guarantees that all I/O operations are performed on the same node.	Any relative path name or full path name.	None. All nodes will participate in parallel I/O.														
MP_MSG_ENVELOPE_BUF -msg_envelope_buf	The size of the message envelope buffer (that is, uncompleted send and receive descriptors).	Any positive number. There is no upper limit, but any value less than 1 MB is ignored.	8 MB														
MP_POLLING_INTERVAL -polling_interval	To change the polling interval (in microseconds).	An integer between 1 and 2 billion	400000														

Table 19. POE environment variables and command-line flags for Message Passing Interface (MPI) (continued)

Environment Variable Command-Line Flag	Set:	Possible Values:	Default:
MP_RETRANSMIT_INTERVAL -retransmit_interval	MP_RETRANSMIT_INTERVAL= <i>nnnn</i> and its command line equivalent, -retransmit_interval= <i>nnnn</i> , control how often the communication subsystem library checks to see if it should retransmit packets that have not been acknowledged. The value <i>nnnn</i> is the number of polling loops between checks.	The acceptable range is from 1000 to INT_MAX	10000 (IP) 400000 (User Space)
MP_LAPI_TRACE_LEVEL	Used in conjunction with AIX tracing for debug purposes. Levels 0-5 are supported.	Levels 0-5	0
MP_USE_BULK_XFER -use_bulk_xfer	<p>Exploits the high performance switch data transfer mechanism. In other environments, this variable does not have any meaning and is ignored.</p> <p>Before you can use <b>MP_USE_BULK_XFER</b>, the system administrator must first enable Remote Direct Memory Access (RDMA). For more information, see <i>IBM Parallel Environment for AIX: Installation</i>. In other environments, this variable does not have any meaning and is ignored.</p> <p>Note that when you use this environment variable, you also need to consider the value of the <b>MP_BULK_MIN_MSG_SIZE</b> environment variable. Messages with lengths that are greater than the value specified <b>MP_BULK_MIN_MSG_SIZE</b> will use the bulk transfer path, if it is available. For more information, see the entry for <b>MP_BULK_MIN_MSG_SIZE</b> in this table.</p>	yes no	no

Table 19. POE environment variables and command-line flags for Message Passing Interface (MPI) (continued)

Environment Variable Command-Line Flag	Set:	Possible Values:	Default:
MP_BULK_MIN_MSG_SIZE -bulk_min_msg_size	Contiguous messages with data lengths greater than or equal to the value you specify for this environment variable will use the bulk transfer path, if it is available. Messages with data lengths that are smaller than the value you specify for this environment variable, or are noncontiguous, will use packet mode transfer.	The acceptable range is from 4096 to 2147483647 (INT_MAX).  <b>Possible values:</b> nnnnn (byte) nnnK (where: K = 1024 bytes) nnM (where: M = 1024*1024 bytes) nnG (where: G = 1 billion bytes)	153600
MP_SHARED_MEMORY -shared_memory	To specify the use of shared memory (instead of IP or the high performance switch) for message passing between tasks running on the same node. <b>Note:</b> In past releases, the <b>MP_SHM_CC</b> environment variable was used to enable or disable the use of shared memory for certain 64-bit MPI collective communication operations. Beginning with the PE 4.2 release, this environment variable has been removed. You should now use <b>MP_SHARED_MEMORY</b> to enable shared memory for both collective communication and point-to-point routines. The default setting for <b>MP_SHARED_MEMORY</b> is <b>yes</b> (enable shared memory).	<b>yes</b> <b>no</b>	<b>yes</b>
MP_SINGLE_THREAD -single_thread	To avoid lock overheads in a program that is known to be single-threaded. MPE_I non-blocking collective, MPI-IO and MPI one-sided are unavailable if this variable is set to <b>yes</b> . Results are undefined if this variable is set to <b>yes</b> with multiple application message passing threads in use.	<b>yes</b> <b>no</b>	<b>no</b>
MP_THREAD_STACKSIZE -thread_stacksize	To specify the additional stack size allocated for user subroutines running on an MPI service thread. If you do not allocate enough space, the program may encounter a SIGSEGV exception or more subtle failures.	nnnnn nnnK (where: K = 1024 bytes) nnM (where: M = 1024*1024 bytes)	0

Table 19. POE environment variables and command-line flags for Message Passing Interface (MPI) (continued)

Environment Variable Command-Line Flag	Set:	Possible Values:	Default:
MP_TIMEOUT None	To change the length of time (in seconds) the communication subsystem will wait for a connection to be established during message-passing initialization.  If the SP security method is "dce and compatibility", you may need to increase the <b>MP_TIMEOUT</b> value to allow POE to wait for the DCE servers to respond (or timeout if the servers are down).	An integer greater than 0	150
MP_UDP_PACKET_SIZE -udp_packet_size	Allows the user to control the packet size. Specify a positive integer.	A positive integer	Switch 64k, otherwise 8k
MP_WAIT_MODE -wait_mode	<b>Set:</b> To specify how a thread or task behaves when it discovers it is blocked, waiting for a message to arrive.	<b>nopoll</b> <b>poll</b> <b>sleep</b> <b>yield</b>	<b>poll</b> (for User Space and IP)
MP_IO_BUFFER_SIZE -io_buffer_size	To specify the default size of the data buffer used by MPI-IO agents.	An integer less than or equal to 128 MB, in one of these formats: <i>nnnnn</i> <i>nnnK</i> (where K=1024 bytes) <i>nnnM</i> (where M=1024*1024 bytes)	The number of bytes that corresponds to 16 file blocks.
MP_IO_ERRLOG -io_errlog	To specify whether or not to turn on I/O error logging.	<b>yes</b> <b>no</b>	<b>no</b>
MP_REXMIT_BUF_SIZE -rexmit_buf_size	The maximum LAPI level message size that will be buffered locally, to more quickly free up the user send buffer. This sets the size of the local buffers that will be allocated to store such messages, and will impact memory usage, while potentially improving performance. The MPI application message size supported is smaller by, at most, 32 bytes.	<i>nnn</i> bytes (where: <i>nnn</i> > 0 bytes)	16352 bytes
MP_REXMIT_BUF_CNT -rexmit_buf_cnt	The number of retransmit buffers that will be allocated per task. Each buffer is of size <b>MP_REXMIT_BUF_SIZE</b> * <b>MP_REXMIT_BUF_CNT</b> . This count controls the number of in-flight messages that can be buffered to allow prompt return of application send buffers.	<i>nnn</i> (where: <i>nnn</i> > 0)	128

---

## MP\_BUFFER\_MEM details

### Set:

To control the amount of memory PE MPI allows for the buffering of early arrival message data. Message data that is sent without knowing if the receive is posted is said to be sent **eagerly**. If the message data arrives before the receive is posted, this is called an **early arrival** and must be buffered at the receive side.

There are two way this environment variable can be used:

1. To specify the pool size for memory to be allocated at MPI initialization time and dedicated to buffering of early arrivals. Management of pool memory for each early arrival is fast, which helps performance, but memory that is set aside in this pool is not available for other uses. Eager sending is throttled by PE MPI to be certain there will never be an early arrival that cannot fit **within the pool**. (To **throttle** a car engine is to choke off its air and fuel intake by lifting your foot from the gas pedal when you want to keep the car from going faster than you can control).
2. To specify the pool size for memory to be allocated at MPI initialization time and, with a second argument, an upper bound of memory to be used if the pre-allocated pool is not sufficient. Eager sending is throttled to be certain there will never be an early arrival that cannot fit **within the upper bound**. Any early arrival will be stored in the pre-allocated pool using its faster memory management if there is room, but if not, malloc and free will be used.

The constraints on eager send must be pessimistic because they must guarantee an early arrival buffer no matter how the application behaves. Real applications at large task counts may suffer performance loss due to pessimistic throttling of eager sending, even though the application has only a modest need for early arrival buffering.

Setting a higher bound allows more and larger messages to be sent eagerly. If the application is well behaved, it is likely that the pre-allocated pool will supply all the buffer space needed. If not, malloc and free will be used but never beyond the stated upper bound.

### Possible values:

*nnnnn* (byte)  
*nnnK* (where: K = 1024 bytes)  
*nnM* (where: M = 1024\*1024 bytes)  
*nnG* (where: G = 1 billion bytes)

### Formats:

**M1**  
**M1,M2**  
**,M2** (a comma followed by the **M2** value)

**M1** specifies the size of the pool to be allocated at initialization time. **M1** must be between 0 and 256 MB.

**M2** specifies the upper bound of memory that PE MPI will allow to be used for early arrival buffering in the most extreme case of sends without waiting receives. PE MPI will throttle senders back to rendezvous protocol (stop trying to use eager send) before allowing the early arrivals at a receive side to overflow the upper bound.

There is no limit enforced on the value you can specify for **M2**, but be aware that a program that does not behave as expected has the potential to malloc this much memory, and terminate if it is not available.

When **MP\_BUFFER\_MEM** is allowed to default, or is specified with a single argument, **M1**, the upper bound is set to the pool size, and eager sending will be throttled soon enough at each sender to ensure that the buffer pool cannot overflow at any receive side. If **M2** is smaller than **M1**, **M2** is ignored.

The format that omits **M1** is used to tell PE MPI to use its default size pre-allocated pool, but set the upper bound as specified with **M2**. This removes the need for a user to remember the default **M1** value when the intention is to only change the **M2** value.

It is expected that only jobs with hundreds of task will have any need to set **M2**. For most of these jobs, there will be an **M1,M2** setting that eliminates the need for PE MPI to throttle eager sends, while allowing all early arrivals that the application actually creates to be buffered within the pre-allocated pool.

Table 20. POE environment variables/command-line flags for corefile generation

The Environment Variable/Command-Line Flag(s):	Set:	Possible Values:	Default:
MP_COREDIR -coredir	Creates a separate directory for each task's core file.	Any valid directory name, or "none" to bypass creating a new directory.	<i>coredir.taskid</i>
MP_COREFILE_FORMAT -corefile_format	The format of corefiles generated when processes terminate abnormally.	The string "STDERR" (to specify that the lightweight corefile information should be written to standard error) or any other string (to specify the lightweight corefile name).	If not set/specified, standard AIX corefiles will be generated.
MP_COREFILE_SIGTERM -corefile_sigterm	Determines if POE should generate a core file when a <b>SIGTERM</b> signal is received. Valid values are <b>yes</b> and <b>no</b> . If not set, the default is <b>no</b> .	<b>yes, no</b>	<b>no</b>

Table 21. Other POE environment variables/command-line flags

The Environment Variable/Command-Line Flag(s):	Set:	Possible Values:	Default:
MP_DBXPROMPTMOD (no associated command-line flag)	A modified <b>dbx</b> prompt. The <b>dbx</b> prompt <code>\n(dbx)</code> is used by the <b>pdbx</b> command as an indicator denoting that a <b>dbx</b> subcommand has completed. This environment variable modifies that prompt. Any value assigned to it will have a "." prepended and will then be inserted in the <code>\n(dbx)</code> prompt between the "x" and the ")". This environment variable is useful when the string <code>\n(dbx)</code> is present in the output of the program being debugged.	Any string.	None

Table 21. Other POE environment variables/command-line flags (continued)

The Environment Variable/Command-Line Flag(s):	Set:	Possible Values:	Default:
MP_EUIDEVELOP -euiddevelop	Controls the level of parameter checking during execution. Setting this to <b>yes</b> enables some intertask parameter checking which may help uncover certain problems, but slows execution. Normal mode does only relatively inexpensive, local parameter checking. Minimum allows PE MPI to bypass even local parameter checking on certain performance critical calls.	<i>yes</i> (for “develop”), <i>no</i> or <i>nor</i> (for “normal”), <i>deb</i> (for “debug”) and <i>min</i> (for “minimum”).	<i>no</i>
MP_STATISTICS -statistics	Provides the ability to gather communication statistics for User Space jobs.	<b>yes</b> <b>no</b> <b>print</b>	<i>no</i>
MP_FENCE (no associated command-line flag)	A “fence” character string for separating arguments you want parsed by POE from those you do not.	Any string.	None
MP_NOARGLIST (no associated command-line flag)	Whether or not POE ignores the argument list. If set to <i>yes</i> , POE will not attempt to remove POE command-line flags before passing the argument list to the user’s program.	<i>yes no</i>	<i>no</i>
MP_PRIORITY (no associated command-line flag)	A dispatch priority class for execution or a string of high/low priority values. See <i>IBM Parallel Environment for AIX: Installation</i> for more information on dispatch priority classes.	Any of the dispatch priority classes set up by the system administrator or a string of high/low priority values in the file <i>/etc/poe.priority</i> .	None
MP_PRINTENV -printenv	Whether to produce a report of the current settings of MPI environment variables, across all tasks in a job. If <b>yes</b> is specified, the MPI environment variable information is gathered at initialization time from all tasks, and forwarded to task 0, where the report is prepared. If a <i>script_name</i> is specified, the script is run on each node, and the output script is forwarded to task 0 and included in the report.  When a variable’s value is the same for all tasks, it is printed only once. If it is different for some tasks, an asterisk (*) appears in the report after the word “Task”.	<b>no</b> Do not produce a report of MPI environment variable settings.  <b>yes</b> Produce a report of MPI environment variable settings.  <i>script_name</i> Produce the report (same as <b>yes</b> ), then run the script specified here.	<b>no</b>

Table 21. Other POE environment variables/command-line flags (continued)

The Environment Variable/Command-Line Flag(s):	Set:	Possible Values:	Default:
MP_UTE	To include the UTE (Unified Trace Environment) library in the link step, allowing the user to collect data from the application using PE Benchmark. For more information, see <i>IBM Parallel Environment for AIX: Operation and Use, Volume 2</i> .	<b>yes</b> Include the UTE library in the link step. <b>no</b> Do not include the UTE library in the link step.	no



## Chapter 12. Parallel utility subroutines

This chapter includes descriptions of the parallel utility subroutines that are available for parallel programming. These user-callable, threadsafe subroutines take advantage of the parallel operating environment (POE).

Table 22. Parallel utility subroutines

Subroutine name	Purpose
"mpc_isatty" on page 89	Determines whether a device is a terminal on the home node.
"MP_BANDWIDTH, mpc_bandwidth" on page 91	Obtains user space switch bandwidth statistics.
"MP_DISABLEINTR, mpc_disableintr" on page 96	Disables message arrival interrupts on a node.
"MP_ENABLEINTR, mpc_enableintr" on page 99	Enables message arrival interrupts on a node.
"MP_FLUSH, mpc_flush" on page 102	Flushes task output buffers.
"MP_INIT_CKPT, mpc_init_ckpt" on page 104	Starts user-initiated checkpointing.
"MP_QUERYINTR, mpc_queryintr" on page 106	Returns the state of interrupts on a node.
"MP_QUERYINTRDELAY, mpc_queryintrdelay" on page 109	The original purpose of this routine was to return the current interrupt delay time. This routine currently returns zero.
"MP_SET_CKPT_CALLBACKS, mpc_set_ckpt_callbacks" on page 110	Registers subroutines to be invoked when the application is checkpointed, resumed, and restarted.
"MP_SETINTRDELAY, mpc_setintrdelay" on page 113	This function formerly set the delay parameter. It now performs no action.
"MP_STATISTICS_WRITE, mpc_statistics_write" on page 114	Print both MPI and LAPI transmission statistics.
"MP_STATISTICS_ZERO, mpc_statistics_zero" on page 117	Resets (zeros) the <b>MPCI_stats_t</b> structure. It has no effect on LAPI.
"MP_STDOUT_MODE, mpc_stdout_mode" on page 118	Sets the mode for STDOUT.
"MP_STDOUTMODE_QUERY, mpc_stdoutmode_query" on page 121	Queries the current STDOUT mode setting.
"MP_UNSET_CKPT_CALLBACKS, mpc_unset_ckpt_callbacks" on page 123	Unregisters checkpoint, resume, and restart application callbacks.
"pe_dbg_breakpoint" on page 125	Provides a communication mechanism between Parallel Operating Environment (POE) and an attached third party debugger (TPD).
"pe_dbg_checkpoint" on page 131	Checkpoints a process that is under debugger control, or a group of processes.
"pe_dbg_checkpoint_wait" on page 135	Waits for a checkpoint, or pending checkpoint file I/O, to complete.
"pe_dbg_getcrd" on page 137	Returns the checkpoint/restart ID.
"pe_dbg_getrtid" on page 138	Returns real thread ID of a thread in a specified process given its virtual thread ID.
"pe_dbg_getvtid" on page 139	Returns virtual thread ID of a thread in a specified process given its real thread ID.
"pe_dbg_read_cr_errfile" on page 140	Opens and reads information from a checkpoint or restart error file.

Table 22. Parallel utility subroutines (continued)

Subroutine name	Purpose
"pe_dbg_restart" on page 141	Restarts processes from a checkpoint file.

---

## mpc\_isatty

### Purpose

Determines whether a device is a terminal on the home node.

### Library

libmpi\_r.a

### C synopsis

```
#include <pm_util.h>
int mpc_isatty(int FileDescriptor);
```

### Description

This parallel utility subroutine determines whether the file descriptor specified by the *FileDescriptor* parameter is associated with a terminal device on the home node. In a parallel operating environment partition, these three file descriptors are implemented as pipes to the partition manager daemon. Therefore, the AIX **isatty()** subroutine will always return **false** for each of them. This subroutine is provided for use by remote tasks that may want to know whether one of these devices is actually a terminal on the home node, for example, to determine whether or not to output a prompt.

### Parameters

#### FileDescriptor

is the file descriptor number of the device. Valid values are:

#### 0 or STDIN

Specifies STDIN as the device to be checked.

#### 1 or STDOUT

Specifies STDOUT as the device to be checked.

#### 2 or STDERR

Specifies STDERR as the device to be checked.

### Notes

This subroutine has a C version only. Also, it is thread safe.

### Return values

In C and C++ calls, the following applies:

- 0 Indicates that the device is *not* associated with a terminal on the home node.
- 1 Indicates that the device *is* associated with a terminal on the home node.
- 1 Indicates an invalid *FileDescriptor* parameter.

### Examples

#### C Example

```
/*
 * Running this program, after compiling with mpcc_r,
 * without redirecting STDIN, produces the following output:
 *
 *      isatty() reports STDIN as a non-terminal device
```

## mpc\_isatty

```
*      mpc_isatty() reports STDIN as a terminal device
*/

#include "pm_util.h"

main()
{
    if (isatty(STDIN)) {
        printf("isatty() reports STDIN as a terminal device\n");
    } else {
        printf("isatty() reports STDIN as a non-terminal device\n");
        if (mpc_isatty(STDIN)) {
            printf("mpc_isatty() reports STDIN as a terminal device\n");
        } else {
            printf("mpc_isatty() reports STDIN as a non-terminal device\n");
        }
    }
}
```

## MP\_BANDWIDTH, mpc\_bandwidth

### Purpose

Obtains user space switch bandwidth statistics.

### Library

libmpi\_r.a

### C synopsis

```
#include <pm_util.h>
#include <lapi.h>
int mpc_bandwidth(lapi_handle_t hndl, int flag, bw_stat_t *bw);
```

### FORTTRAN synopsis

```
MP_BANDWIDTH(INTEGER HNDL, INTEGER FLAG, INTEGER*8 BW_SENT, INTEGER*8 BW_RECV,
  INTEGER*8 BW_TIME_SEC, INTEGER*4 BW_TIME_USEC, INTEGER RC)
```

### Description

This parallel utility subroutine is a wrapper API program that users can call to obtain the user space switch bandwidth statistics. LAPI's Query interface is used to obtain byte counts of the data sent and received. This routine returns the byte counts and time values to allow the bandwidth to be calculated.

For C and C++ language programs, this routine uses a structure that contains the data count fields, as well as time values in both seconds and microseconds. These are filled in at the time of the call, from the data obtained by the LAPI Query interface and a "get time of day" call.

This routine requires a valid LAPI handle for LAPI programs. For MPI programs, the handle is not required. A flag parameter is required to indicate whether the call has been made from an MPI or LAPI program.

If the program is a LAPI program, the flag `MP_BW_LAPI` must be set and the handle value must be specified. If the program is an MPI program, the flag `MP_BW_MPI` must be set, and any handle specified is ignored.

In the case where a program uses both MPI and LAPI in the same program, where `MP_MSG_API` is set to either `mpi,lapi` or `mpi_lapi`, separate sets of statistics are maintained for the MPI and LAPI portions of the program. To obtain the MPI bandwidth statistics, this routine must be called with the `MP_BW_MPI` flag, and any handle specified is ignored. To obtain the LAPI bandwidth statistics, this routine must be called with the `MP_BW_LAPI` flag and a valid LAPI handle value.

### Parameters

In C, *bw* is a pointer to a `bw_stat_t` structure. This structure is defined as:

```
typedef struct{
    unsigned long long switch_sent;
    unsigned long long switch_recv;
    int64_t time_sec;
    int32_t time_usec;
} bw_stat_t;
```

where:

## MP\_BANDWIDTH

*switch\_sent* is an unsigned long long value of the number of bytes sent.

*switch\_recv* is an unsigned long long value of the number of bytes received.

*time\_sec* is a 64-bit integer value of time in seconds.

*time\_usec* is a 32-bit integer value of time in microseconds.

In FORTRAN:

*BW\_SENT* is a 64-bit integer value of the number of bytes sent.

*BW\_RECV* is a 64-bit integer value of the number of bytes received.

*BW\_TIME\_SEC* is a 64-bit integer time value of time in seconds.

*BW\_TIME\_USEC* is a 32-bit integer time value of time in microseconds.

*Flag* is either MP\_BW\_MPI or MP\_BW\_LAPI, indicating whether the program is using MPI or LAPI.

*Bw\_data* is a pointer to the bandwidth data structure, that will include the timestamp and bandwidth data count of sends and receives as requested. The bandwidth data structure may be declared and passed locally by the calling program.

*Hndl* is a valid LAPI handle filled in by a LAPI\_Init() call for LAPI programs. For MPI programs, this is ignored.

*RC* in FORTRAN, will contain an integer value returned by this function. This should always be the last parameter.

## Notes

1. The send and receive data counts are for bandwidth data at the software level of current tasks running, and not what the adapter is capable of.
2. Intranode communication using shared memory will specifically *not* be measured with this API. Likewise, this API does not return values of the bandwidth of local data sent to itself.
3. In the case with striping over multiple adapters, the data counts are an aggregate of the data exchanged at the application level, and not on a per-adapter basis.

## Return values

**0** Indicates successful completion.

**-1** Incorrect flag (not MP\_BW\_MPI or MP\_BW\_LAPI).

**greater than 0**  
See the list of LAPI error codes in *IBM RSCT: LAPI Programming Guide*.

## Examples

### C Examples

1. To determine the bandwidth in an MPI program:

```
#include <mpi.h>
#include <time.h>
#include <lapi.h>
```

```

#include <pm_util.h>
int rc;
main(int argc, char *argv[])
{
    bw_stat_t bw_in;
    MPI_Init(&argc, &argv);
    .
    .
    .
    /* start collecting bandwidth .. */
    rc = mpc_bandwidth(NULL, MP_BW_MPI, &bw_in);
    .
    .
    .
    printf("Return from mpc_bandwidth ...rc = %d.\n",rc);
    printf("Bandwidth of data sent: %lld.\n",
        bw_in.switch_sent);
    printf("Bandwidth of data recv: %lld.\n",
        bw_in.switch_recv);
    printf("time(seconds): %lld.\n",bw_in.time_sec);
    printf("time(mseconds): %d.\n",bw_in->time_usec);
    .
    .
    .
    MPI_Finalize();
    exit(rc);
}

```

## 2. To determine the bandwidth in a LAPI program:

```

#include <lapi.h>
#include <time.h>
#include <pm_util.h>
int rc;
main(int argc, char *argv[])
{
    lapi_handle_t hndl;
    lapi_info_t info;
    bw_stat_t work;
    bw_stat_t bw_in;
    bzero(&info, sizeof(lapi_info_t));
    rc = LAPI_Init(&hndl, &info);
    .
    .
    .
    rc = mpc_bandwidth(hndl, MP_BW_LAPI, &bw_in);
    .
    .
    .
    printf("Return from mpc_bandwidth ...rc = %d.\n",rc);
    printf("Bandwidth of data sent: %lld.\n",
        bw_in.switch_sent);
    printf("Bandwidth of data recv: %lld.\n",
        bw_in.switch_recv);
    printf("time(seconds): %lld.\n", bw_in.time_sec);
    printf("time(mseconds): %d.\n",bw_in.time_usec);
    .
    .
    .
    LAPI_Term(hndl);
    exit(rc);
}

```

## FORTTRAN Examples

### 1. To determine the bandwidth in an MPI program:

## MP\_BANDWIDTH

```
program bw_mpi
include "mpif.h"
include "lapif.h"
integer retcode
integer taskid
integer numtask
integer hndl
integer*8 bw_secs
integer*4 bw_usecs
integer*8 bw_sent_data
integer*8 bw_recv_data
.
.
.
call mpi_init(retcode)
call mpi_comm_rank(mpi_comm_world, taskid, retcode)
write (6,*) 'Taskid is ',taskid
.
.
.
call mp_bandwidth(hndl,MP_BW_MPI, bw_sent_data, bw_recv_data, bw_secs,
bw_usecs,retcode)
write (6,*) 'MPI_BANDWIDTH returned. Time (sec) is ',bw_secs
write (6,*) ' Time (usec) is ',bw_usecs
write (6,*) ' Data sent (bytes): ',bw_sent_data
write (6,*) ' Data received (bytes): ',bw_sent_recv
write (6,*) ' Return code: ',retcode
.
.
.
call mpi_barrier(mpi_comm_world,retcode)
call mpi_finalize(retcode)
```

### 2. To determine the bandwidth in a LAPI program:

```
program bw_lapi
include "mpif.h"
include "lapif.h"
TYPE (LAPI_INFO_T) :: lapi_info
integer retcode
integer taskid
integer numtask
integer hndl
integer*8 bw_secs
integer*4 bw_usecs
integer*8 bw_sent_data
integer*8 bw_recv_data
.
.
.
call lapi_init(hndl, lapi_info, retcode)
.
.
.
call mp_bandwidth(hndl,MP_BW_LAPI, bw_sent_data, bw_recv_data, bw_secs,
bw_usecs,retcode)
write (6,*) 'MPI_BANDWIDTH returned. Time (sec) is ',bw_secs
write (6,*) ' Time (usec) is ',bw_usecs
write (6,*) ' Data sent (bytes): ',bw_sent_data
write (6,*) ' Data received (bytes): ',bw_sent_recv
write (6,*) ' Return code: ',retcode
.
.
.
call lapi_term(hndl,retcode)
```

## Related information

Commands:

- mpcc\_r
- mpCC\_r
- mpxlf\_r
- mpxlf90\_r
- mpxlf95\_r

Subroutines:

- MP\_STATISTICS\_WRITE, mpc\_statistics\_write
- MP\_STATISTICS\_ZERO, mpc\_statistics\_zero

---

## MP\_DISABLEINTR, mpc\_disableintr

### Purpose

Disables message arrival interrupts on a node.

### Library

libmpi\_r.a

### C synopsis

```
#include <pm_util.h>
int mpc_disableintr();
```

### FORTRAN synopsis

```
MP_DISABLEINTR(INTEGER RC)
```

### Description

This parallel utility subroutine disables message arrival interrupts on the individual node on which it is run. Use this subroutine to dynamically control masking interrupts on a node.

### Parameters

In FORTRAN, *RC* will contain one of the values listed under **Return Values**.

### Notes

- This subroutine is only effective when the communication subsystem is active. This is from MPI\_INIT to MPI\_FINALIZE. If this subroutine is called when the subsystem is inactive, the call will have no effect and the return code will be -1.
- This subroutine overrides the setting of the environment variable MP\_CSS\_INTERRUPT.
- Inappropriate use of the interrupt control subroutines may reduce performance.
- This subroutine can be used for IP and User Space protocols.
- This subroutine is thread-safe.
- Using this subroutine will suppress the MPI-directed switching of interrupt mode, leaving the user in control for the rest of the run. See MPI\_FILE\_OPEN and MPI\_WIN\_CREATE in *IBM Parallel Environment for AIX: MPI Subroutine Reference*.

### Return values

- |    |  |
|----|--|
| 0  | Indicates successful completion.   |
| -1 | Indicates that the MPI library was not active. The call was either made before MPI_INIT or after MPI_FINALIZE. |

### Examples

#### C Example

```
/*
 * Running this program, after compiling with mpcc_r,
 * without setting the MP_CSS_INTERRUPT environment variable,
 * and without using the "-css_interrupt" command-line option,
 * produces the following output:
```

```

*
*   Interrupts are DISABLED
*   About to enable interrupts..
*   Interrupts are ENABLED
*   About to disable interrupts...
*   Interrupts are DISABLED
*/

#include "pm_util.h"

#define QUERY if (intr = mpc_queryintr()) {\
    printf("Interrupts are ENABLED\n");\
} else {\
    printf("Interrupts are DISABLED\n");\
}

main()
{
    int intr;

    QUERY

    printf("About to enable interrupts...\n");
    mpc_enableintr();

    QUERY

    printf("About to disable interrupts...\n");
    mpc_disableintr();

    QUERY
}

```

### FORTRAN Example

Running the following program, after compiling with **mpxlf\_r**, without setting the **MP\_CSS\_INTERRUPT** environment variable, and without using the **-css\_interrupt** command-line option, produces the following output:

```

Interrupts are DISABLED
About to enable interrupts..
Interrupts are ENABLED
About to disable interrupts...
Interrupts are DISABLED

PROGRAM INTR_EXAMPLE

INTEGER RC

CALL MP_QUERYINTR(RC)
IF (RC .EQ. 0) THEN
    WRITE(6,*)'Interrupts are DISABLED'
ELSE
    WRITE(6,*)'Interrupts are ENABLED'
ENDIF

WRITE(6,*)'About to enable interrupts...'
CALL MP_ENABLEINTR(RC)

CALL MP_QUERYINTR(RC)
IF (RC .EQ. 0) THEN
    WRITE(6,*)'Interrupts are DISABLED'
ELSE
    WRITE(6,*)'Interrupts are ENABLED'
ENDIF

```

## MP\_DISABLEINTR

```
WRITE(6,*)'About to disable interrupts...'  
CALL MP_DISABLEINTR(RC)  
  
CALL MP_QUERYINTR(RC)  
IF (RC .EQ. 0) THEN  
    WRITE(6,*)'Interrupts are DISABLED'  
ELSE  
    WRITE(6,*)'Interrupts are ENABLED'  
ENDIF  
  
STOP  
END
```

## Related information

Subroutines:

- MP\_ENABLEINTR, mpc\_enableintr
- MP\_QUERYINTR, mpc\_queryintr

---

## MP\_ENABLEINTR, mpc\_enableintr

### Purpose

Enables message arrival interrupts on a node.

### Library

libmpi\_r.a

### C synopsis

```
#include <pm_util.h>
int mpc_enableintr();
```

### FORTRAN synopsis

```
MP_ENABLEINTR(INTEGER RC)
```

### Description

This parallel utility subroutine enables message arrival interrupts on the individual node on which it is run. Use this subroutine to dynamically control masking interrupts on a node.

### Parameters

In FORTRAN, *RC* will contain one of the values listed under **Return Values**.

### Notes

- This subroutine is only effective when the communication subsystem is active. This is from MPI\_INIT to MPI\_FINALIZE. If this subroutine is called when the subsystem is inactive, the call will have no effect and the return code will be -1.
- This subroutine overrides the setting of the environment variable MP\_CSS\_INTERRUPT.
- Inappropriate use of the interrupt control subroutines may reduce performance.
- This subroutine can be used for IP and User Space protocols.
- This subroutine is thread safe.
- Using this subroutine will suppress the MPI-directed switching of interrupt mode, leaving the user in control for the rest of the run. See MPI\_FILE\_OPEN and MPI\_WIN\_CREATE in *IBM Parallel Environment for AIX: MPI Subroutine Reference*.

### Return values

- |    |  |
|----|--|
| 0  | Indicates successful completion.   |
| -1 | Indicates that the MPI library was not active. The call was either made before MPI_INIT or after MPI_FINALIZE. |

### Examples

#### C Example

```
/*
 * Running this program, after compiling with mpcc_r,
 * without setting the MP_CSS_INTERRUPT environment variable,
 * and without using the "-css_interrupt" command-line option,
 * produces the following output:
```

## MP\_ENABLEINTR

```
*
*   Interrupts are DISABLED
*   About to enable interrupts..
*   Interrupts are ENABLED
*   About to disable interrupts...
*   Interrupts are DISABLED
*/

#include "pm_util.h"

#define QUERY if (intr = mpc_queryintr()) {\
    printf("Interrupts are ENABLED\n");\
} else {\
    printf("Interrupts are DISABLED\n");\
}

main()
{
    int intr;

    QUERY

    printf("About to enable interrupts...\n");
    mpc_enableintr();

    QUERY

    printf("About to disable interrupts...\n");
    mpc_disableintr();

    QUERY
}
```

### FORTRAN Example

Running this program, after compiling with **mpxlf\_r**, without setting the **MP\_CSS\_INTERRUPT** environment variable, and without using the **-css\_interrupt** command-line option, produces the following output:

```
Interrupts are DISABLED
About to enable interrupts..
Interrupts are ENABLED
About to disable interrupts...
Interrupts are DISABLED

PROGRAM INTR_EXAMPLE

INTEGER RC

CALL MP_QUERYINTR(RC)
IF (RC .EQ. 0) THEN
    WRITE(6,*)'Interrupts are DISABLED'
ELSE
    WRITE(6,*)'Interrupts are ENABLED'
ENDIF

WRITE(6,*)'About to enable interrupts...'
CALL MP_ENABLEINTR(RC)

CALL MP_QUERYINTR(RC)
IF (RC .EQ. 0) THEN
    WRITE(6,*)'Interrupts are DISABLED'
ELSE
    WRITE(6,*)'Interrupts are ENABLED'
ENDIF
```

```

WRITE(6,*)'About to disable interrupts...'
CALL MP_DISABLEINTR(RC)

CALL MP_QUERYINTR(RC)
IF (RC .EQ. 0) THEN
    WRITE(6,*)'Interrupts are DISABLED'
ELSE
    WRITE(6,*)'Interrupts are ENABLED'
ENDIF

STOP
END

```

## Related information

Subroutines:

- MP\_DISABLEINTR, mpc\_disableintr
- MP\_QUERYINTR, mpc\_queryintr

---

## MP\_FLUSH, mpc\_flush

### Purpose

Flushes task output buffers.

### Library

libmpi\_r.a

### C synopsis

```
#include <pm_util.h>
int mpc_flush(int option);
```

### FORTRAN synopsis

MP\_FLUSH(*INTEGER OPTION*)

### Description

This parallel utility subroutine flushes output buffers from all of the parallel tasks to STDOUT at the home node. This is a synchronizing call across all parallel tasks.

If the current STDOUT mode is ordered, then when all tasks have issued this call or when any of the output buffers are full:

1. All STDOUT buffers are flushed and put out to the user screen (or redirected) in task order.
2. An acknowledgement is sent to all tasks and control is returned to the user.

If current STDOUT mode is unordered and all tasks have issued this call, all output buffers are flushed and put out to the user screen (or redirected).

If the current STDOUT mode is single and all tasks have issued this call, the output buffer for the current single task is flushed and put out to the user screen (or redirected).

### Parameters

**option** is an AIX file descriptor. The only valid value is:

- |   |                                    |
|---|------------------------------------|
| 1 | Indicates to flush STDOUT buffers. |
|---|------------------------------------|

### Notes

- This is a synchronizing call regardless of the current STDOUT mode.
- All STDOUT buffers are flushed at the end of the parallel job.
- If mpc\_flush is not used, standard output streams not terminated with a new-line character are buffered, even if a subsequent read to standard input is made. This may cause prompt message to appear only after input has been read.
- This subroutine is thread safe.

### Return values

In C and C++ calls, the following applies:

- |   |                                 |
|---|---------------------------------|
| 0 | Indicates successful completion |
|---|---------------------------------|

- 1 Indicates that an error occurred. A message describing the error will be issued.

## Examples

### C Example

The following program uses **poe** with the **-labelio yes** option and three tasks:

```
#include <pm_util.h>

main()
{
    mpc_stdout_mode(STDIO_ORDERED);
    printf("These lines will appear in task order\n");
    /*
     * Call mpc_flush here to make sure that one task
     * doesn't change the mode before all tasks have
     * sent the previous printf string to the home node.
     */
    mpc_flush(1);
    mpc_stdout_mode(STDIO_UNORDERED);
    printf("These lines will appear in the order received by the home node\n");
    /*
     * Since synchronization is not used here, one task could actually
     * execute the next statement before one of the other tasks has
     * executed the previous statement, causing one of the unordered
     * lines not to print.
     */
    mpc_stdout_mode(1);
    printf("Only 1 copy of this line will appear from task 1\n");
}
```

Running this C program produces the following output (the task order of lines 4 through 6 may differ):

- 0 : These lines will appear in task order.
- 1 : These lines will appear in task order.
- 2 : These lines will appear in task order.
- 1 : These lines will appear in the order received by the home node.
- 2 : These lines will appear in the order received by the home node.
- 0 : These lines will appear in the order received by the home node.
- 1 : Only 1 copy of this line will appear from task 1.

### FORTRAN Example

```
CALL MP_STDOUT_MODE(-2)
WRITE(6, *) 'These lines will appear in task order'
CALL MP_FLUSH(1)
CALL MP_STDOUT_MODE(-3)
WRITE(6, *) 'These lines will appear in the order received by the home node'
CALL MP_STDOUT_MODE(1)
WRITE(6, *) 'Only 1 copy of this line will appear from task 1'
END
```

## Related information

Subroutines:

- MP\_STDOUT\_MODE, mpc\_stdout\_mode
- MP\_STDOUTMODE\_QUERY, mpc\_stdoutmode\_query

---

## MP\_INIT\_CKPT, mpc\_init\_ckpt

### Purpose

Starts user-initiated checkpointing.

### Library

libmpi\_r.a

### C synopsis

```
#include <pm_ckpt.h>
int mpc_init_ckpt(int flags);
```

### FORTRAN synopsis

```
i = MP_INIT_CKPT(%val(j))
```

### Description

MP\_INIT\_CKPT starts complete or partial user-initiated checkpointing. The checkpoint file name consists of the base name provided by the MP\_CKPTFILE and MP\_CKPTDIR environment variables, with a suffix of the task ID and a numeric checkpoint tag to differentiate it from an earlier checkpoint file.

If the MP\_CKPTFILE environment variable is not specified, a default base name is constructed: **poe.ckpt.tag**, where *tag* is an integer that allows multiple versions of checkpoint files to exist. The file name specified by MP\_CKPTFILE may include the full path of where the checkpoint files will reside, in which case the MP\_CKPTDIR variable is to be ignored. If MP\_CKPTDIR is not defined and MP\_CKPTFILE does not specify a full path name, MP\_CKPTFILE is used as a relative path name from the original working directory of the task.

### Parameters

In C, *flags* can be set to MP\_CUSER, which indicates complete user-initiated checkpointing, or MP\_PUSER, which indicates partial user-initiated checkpointing.

In FORTRAN, *j* should be set to 0 (the value of MP\_CUSER) or 1 (the value of MP\_PUSER).

### Notes

Complete user-initiated checkpointing is a synchronous operation. All tasks of the parallel program must call MP\_INIT\_CKPT. MP\_INIT\_CKPT suspends the calling thread until all other tasks have called it (MP\_INIT\_CKPT). Other threads in the task are not suspended. After all tasks of the application have issued MP\_INIT\_CKPT, a local checkpoint is taken of each task.

In partial user-initiated checkpointing, one task of the parallel program calls MP\_INIT\_CKPT, thus invoking a checkpoint on the entire application. A checkpoint is performed asynchronously on all other tasks. The thread that called MP\_INIT\_CKPT is suspended until the checkpoint is taken. Other threads in the task are not suspended.

Upon returning from the MP\_INIT\_CKPT call, the application continues to run. It may, however, be a restarted application that is now running, rather than the original, if the program was restarted from a checkpoint file.

In a case where several threads in a task call MP\_INIT\_CKPT using the same flag, the calls are serialized.

The task that calls MP\_INIT\_CKPT does not need to be an MPI program.

There are certain limitations associated with checkpointing an application. See “Checkpoint and restart limitations” on page 39 for more information.

For general information on checkpointing and restarting programs, see *IBM Parallel Environment for AIX: Operation and Use, Volume 1*.

For more information on the use of LoadLeveler and checkpointing, see *IBM LoadLeveler for AIX 5L: Using and Administering*.

## Return values

- |    |  |
|----|--|
| 0  | Indicates successful completion.   |
| 1  | Indicates that a restart operation occurred.                                     |
| -1 | Indicates that an error occurred. A message describing the error will be issued. |

## Examples

### C Example

```
#include <pm_ckpt.h>
int mpc_init_ckpt(int flags);
```

### FORTRAN Example

```
i = MP_INIT_CKPT(%val(j))
```

## Related information

Commands:

- poeckpt
- poerestart

Subroutines:

- MP\_SET\_CKPT\_CALLBACKS, mpc\_set\_ckpt\_callbacks
- MP\_UNSET\_CKPT\_CALLBACKS, mpc\_unset\_ckpt\_callbacks

---

## MP\_QUERYINTR, mpc\_queryintr

### Purpose

Returns the state of interrupts on a node.

### Library

libmpi\_r.a

### C synopsis

```
#include <pm_util.h>
int mpc_queryintr();
```

### FORTRAN synopsis

```
MP_QUERYINTR(INTEGER RC)
```

### Description

This parallel utility subroutine returns the state of interrupts on a node.

### Parameters

In FORTRAN, *RC* will contain one of the values listed under **Return Values**.

### Notes

This subroutine is thread safe.

### Return values

- 0       Indicates that interrupts are disabled on the node from which this subroutine is called.
- 1       Indicates that interrupts are enabled on the node from which this subroutine is called.

### Examples

#### C Example

```
/*
 * Running this program, after compiling with mpcc_r,
 * without setting the MP_CSS_INTERRUPT environment variable,
 * and without using the "-css_interrupt" command-line option,
 * produces the following output:
 *
 *   Interrupts are DISABLED
 *   About to enable interrupts..
 *   Interrupts are ENABLED
 *   About to disable interrupts...
 *   Interrupts are DISABLED
 */

#include "pm_util.h"

#define QUERY if (intr = mpc_queryintr()) {\
    printf("Interrupts are ENABLED\n");\
} else {\
    printf("Interrupts are DISABLED\n");\
}
```

```

main()
{
    int intr;

    QUERY

    printf("About to enable interrupts...\n");
    mpc_enableintr();

    QUERY

    printf("About to disable interrupts...\n");
    mpc_disableintr();

    QUERY
}

```

### FORTRAN Example

Running this program, after compiling with **mpxlf\_r**, without setting the **MP\_CSS\_INTERRUPT** environment variable, and without using the **-css\_interrupt** command-line option, produces the following output:

```

Interrupts are DISABLED
About to enable interrupts..
Interrupts are ENABLED
About to disable interrupts...
Interrupts are DISABLED

```

```

PROGRAM INTR_EXAMPLE

INTEGER RC

CALL MP_QUERYINTR(RC)
IF (RC .EQ. 0) THEN
    WRITE(6,*)'Interrupts are DISABLED'
ELSE
    WRITE(6,*)'Interrupts are ENABLED'
ENDIF

WRITE(6,*)'About to enable interrupts...'
CALL MP_ENABLEINTR(RC)

CALL MP_QUERYINTR(RC)
IF (RC .EQ. 0) THEN
    WRITE(6,*)'Interrupts are DISABLED'
ELSE
    WRITE(6,*)'Interrupts are ENABLED'
ENDIF

WRITE(6,*)'About to disable interrupts...'
CALL MP_DISABLEINTR(RC)

CALL MP_QUERYINTR(RC)
IF (RC .EQ. 0) THEN
    WRITE(6,*)'Interrupts are DISABLED'
ELSE
    WRITE(6,*)'Interrupts are ENABLED'
ENDIF

STOP
END

```

### Related information

Subroutines:

- MP\_DISABLEINTR, mpc\_disableintr
- MP\_ENABLEINTR, mpc\_enableintr

---

## MP\_QUERYINTRDELAY, mpc\_queryintrdelay

### Purpose

#### Note

This function is no longer supported and its future use is not recommended. The routine remains available for binary compatibility. If invoked, it performs no action and always returns zero. Applications that include calls to this routine should continue to function as before. We suggest that calls to this routine be removed from source code if it becomes convenient to do so.

The original purpose of this routine was to return the current interrupt delay time. This routine currently returns zero.

---

## MP\_SET\_CKPT\_CALLBACKS, mpc\_set\_ckpt\_callbacks

### Purpose

Registers subroutines to be invoked when the application is checkpointed, resumed, and restarted.

### Library

libmpi\_r.a

### C synopsis

```
#include <pm_ckpt.h>
int mpc_set_ckpt_callbacks(callbacks_t *cbs);
```

### FORTTRAN synopsis

```
MP_SET_CKPT_CALLBACKS(EXTERNAL CHECKPOINT_CALLBACK_FUNC,
                      EXTERNAL RESUME_CALLBACK_FUNC,
                      EXTERNAL RESTART_CALLBACK_FUNC,
                      INTEGER RC)
```

### Description

The MP\_SET\_CKPT\_CALLBACKS subroutine is called to register subroutines to be invoked when the application is checkpointed, resumed, and restarted.

### Parameters

In C, *cbs* is a pointer to a **callbacks\_t** structure. The structure is defined as:

```
typedef struct {
void (*checkpoint_callback)(void);
void (*restart_callback)(void);
void (*resume_callback)(void);
} callbacks_t;
```

where:

<i>checkpoint_callback</i>	Points to the subroutine to be called at checkpoint time.
<i>restart_callback</i>	Points to the subroutine to be called at restart time.
<i>resume_callback</i>	Points to the subroutine to be called when an application is resumed after taking a checkpoint.

In FORTRAN:

<i>CHECKPOINT_CALLBACK_FUNC</i>	Specifies the subroutine to be called at checkpoint time.
<i>RESUME_CALLBACK_FUNC</i>	Specifies the subroutine to be called when an application is resumed after taking a checkpoint.
<i>RESTART_CALLBACK_FUNC</i>	Specifies the subroutine to be called at restart time.
<i>RC</i>	Contains one of the values listed under <b>Return Values</b> .

## Notes

In order to ensure their completion, the callback subroutines cannot be dependent on the action of any other thread in the current process, or any process created by the task being checkpointed, because these threads or processes or both may or may not be running while the callback subroutines are executing.

The callback subroutines cannot contain calls to:

1. MP\_SET\_CKPT\_CALLBACKS, MP\_UNSET\_CKPT\_CALLBACKS, mpc\_set\_ckpt\_callbacks, or mpc\_unset\_ckpt\_callbacks.
2. Any MPI or LAPI subroutines

If a call to MP\_SET\_CKPT\_CALLBACKS is issued while a checkpoint is in progress, it is possible that the newly-registered callback may or may not run during this checkpoint.

There are certain limitations associated with checkpointing an application. See “Checkpoint and restart limitations” on page 39 for more information.

For general information on checkpointing and restarting programs, see *IBM Parallel Environment for AIX: Operation and Use, Volume 1*.

For more information on the use of LoadLeveler and checkpointing, see *IBM LoadLeveler for AIX 5L: Using and Administering*.

## Return values

**-1** Indicates that an error occurred. A message describing the error will be issued.

**non-negative integer**

Indicates the handle that is to be used in MP\_UNSET\_CKPT\_CALLBACKS to unregister the subroutines.

## Examples

### C Example

```
#include <pm_ckpt.h>
int ihndl;
callbacks_t cbs;
void foo(void);
void bar(void);
cbs.checkpoint_callback=foo;
cbs.resume_callback=bar;
cbs.restart_callback=bar;
ihndl = mpc_set_ckpt_callbacks(callbacks_t *cbs);
```

### FORTRAN Example

```
SUBROUTINE FOO
  :
  :
  RETURN
END
SUBROUTINE BAR
  :
  :
  RETURN
END
PROGRAM MAIN
  EXTERNAL FOO, BAR
  INTEGER HANDLE, RC
```

## MP\_SET\_CKPT\_CALLBACKS

```

:
:
CALL MP_SET_CKPT_CALLBACKS(FOO,BAR,BAR,HANDLE)
IF (HANDLE .NE. 0) STOP 666
:
:
CALL MP_UNSET_CKPT_CALLBACKS(HANDLE,RC)
:
:
END
```

## Related information

Commands:

- poeckpt
- poerestart

Subroutines:

- MP\_INIT\_CKPT, mpc\_init\_ckpt
- MP\_UNSET\_CKPT\_CALLBACKS, mpc\_unset\_ckpt\_callbacks

---

## MP\_SETINTRDELAY, mpc\_setintrdelay

### Purpose

#### Note

This function is no longer supported and its future use is not recommended. The routine remains available for binary compatibility. If invoked, it performs no action and always returns zero. Applications that include calls to this routine should continue to function as before. We suggest that calls to this routine be removed from source code if it becomes convenient to do so.

This function formerly set the delay parameter. It now performs no action.

---

## MP\_STATISTICS\_WRITE, mpc\_statistics\_write

### Description

Print both MPI and LAPI transmission statistics.

### Library

libmpi\_r.a

### C synopsis

```
#include <pm_util.h>
int mpc_statistics_write(FILE *fp);
```

### FORTRAN synopsis

```
MP_STATISTICS_WRITE(INTEGER FILE_DESCRIPTOR, INTEGER RC)
```

### Description

If the **MP\_STATISTICS** environment variable is set to **yes**, MPI will keep a running total on a set of statistical data. If an application calls this function after **MPI\_INIT** is completed, but before **MPI\_FINALIZE** is called, it will print out the current total of all available MPI and LAPI data. If this function is called after **MPI\_FINALIZE** is completed, it will print out **only** the final MPI data.

**Note:** LAPI will always keep its own statistical total with or without having **MP\_STATISTICS** set.

This function can be added to an MPI program to check communication progress. However, keeping statistical data costs computing cycles, and may impair bandwidth.

In the output, each piece of MPI statistical data is preceded by **MPI**, and each piece of LAPI statistical data is preceded by **LAPI**.

The **MPCI\_stats\_t** structure contains this statistical information, which is printed out:

<b>sends</b>	Count of sends initiated.
<b>sendsComplete</b>	Count of sends completed (message sent).
<b>sendWaitsComplete</b>	Count of send waits completed (blocking and non-blocking).
<b>recvs</b>	Count of receives initiated.
<b>recvWaitsComplete</b>	Count of receive waits complete.
<b>earlyArrivals</b>	Count of messages received for which no receive was posted.
<b>earlyArrivalsMatched</b>	Count of early arrivals for which a posted receive has been found.
<b>lateArrivals</b>	Count of messages received for which a receive was posted.
<b>shoves</b>	Count of calls to lapi_send_msg.
<b>pulls</b>	Count of calls to lapi_recv and lapi_recv_vec.

<b>threadedLockYields</b>	Count of lock releases due to waiting threads.
<b>unorderedMsgs</b>	Count of the total number of out of order messages.
<b>buffer_mem_hwmrk</b>	The peak of the memory usage of <b>buffer_memory</b> for the early arrivals.  If the peak memory usage is greater than the amount preallocated with environment variable <b>MP_BUFFER_MEM</b> , you may wish to increase the preallocation. If the peak memory usage is significantly less than the amount preallocated, you may wish to decrease the preallocation, but set an upper bound that equals the previous preallocation value.
<b>tokenStarveds</b>	Number of times a message with the length less than or equal to eager limit were forced to use rendezvous protocol.  If there are more than a few times a message was forced to use rendezvous protocol, you may wish to increase the upper bound given by the second argument of environment variable <b>MP_BUFFER_MEM</b> .
<b>envelope_mem_used</b>	Number of bytes the memory buffer used for storing the envelopes.

The **lapi\_stats\_t** structure contains this statistical information:

<b>Tot_retrans_pkt_cnt</b>	Retransmit packet count.
<b>Tot_gho_pkt_cnt</b>	Ghost packets count.
<b>Tot_pkt_sent_</b>	Total packets sent.
<b>Tot_pkt_rcv_cnt</b>	Total packets received.
<b>Tot_data_sent</b>	Count of total data sent.
<b>Tot_data_rcv</b>	Count of total data received.

## Parameters

*fp* In C, *fp* is either STDOUT, STDERR or a FILE pointer returned by the fopen function.

In FORTRAN, FILE\_DESCRIPTOR is the AIX file descriptor of the file that this function will write to, having these values:

- 1** Indicates that the output is to be written to STDOUT.
- 2** Indicates that the output is to be written to STDERR.

**Other** Indicates the integer returned by the XL FORTRAN utility **getfd**, if the output is to be written to an application-defined file.

The **getfd** utility converts a FORTRAN LUNIT number to an AIX file descriptor. See **Examples** for more detail.

**RC** In FORTRAN, **RC** will contain the integer value returned by this function. See **Return Values** for more detail.

### Return values

- 1      Neither MPI nor LAPI statistics are available.
- 0      Both MPI and LAPI statistics are available.
- 1      Only MPI statistics are available.
- 2      Only LAPI statistics are available.

### Examples

#### C Example

```
#include "pm_util.h"

.....

MPI_Init( ... );

MPI_Send( ... );

MPI_Recv( ... );

/* Write statistics to standard out */
mpc_statistics_write(stdout);

MPI_Finalize();
```

#### FORTRAN Example

```
integer(4) LUNIT, stat_ofile, stat_rc, getfd

call MPI_INIT (ierror)
.....

c   stat_ofile = 1 if output is to go to stdout
c   stat_ofile = 2 if output is to go to stderr
c   If output is to go a file do the following
LUNIT = 4
OPEN (LUNIT, FILE="/tmp/mpi_stat.out")
CALL FLUSH_(LUNIT)
stat_ofile = getfd(LUNIT)
call MP_STATISTICS_WRITE(stat_ofile, stat_rc)
call MPI_FINALIZE(ierror)

.....
```

---

## MP\_STATISTICS\_ZERO, mpc\_statistics\_zero

### Purpose

Resets (zeros) the **MPCI\_stats\_t** structure. It has no effect on LAPI.

### Library

libmpi\_r.a

### C synopsis

```
#include <pm_util.h>
mpc_statistics_zero();
```

### FORTTRAN synopsis

```
MP_STATISTICS_ZERO()
```

### Description

If the **MP\_STATISTICS** environment variable is set to **yes**, MPI will keep a running total on a set of statistical data, after **MPI\_INIT** is completed. At any time during execution, the application can call this function to reset the current total to zero.

### Parameters

None.

### Return values

None.

---

## MP\_STDOUT\_MODE, mpc\_stdout\_mode

### Purpose

Sets the mode for STDOUT.

### Library

libmpi\_r.a

### C synopsis

```
#include <pm_util.h>
int mpc_stdout_mode(int mode);
```

### FORTTRAN synopsis

```
MP_STDOUT_MODE(INTEGER MODE)
```

### Description

This parallel utility subroutine requests that STDOUT be set to single, ordered, or unordered mode. In single mode, only one task output is displayed. In unordered mode, output is displayed in the order received at the home node. In ordered mode, each parallel task writes output data to its own buffer. When a flush request is made all the task buffers are flushed, in order of task ID, to STDOUT home node.

### Parameters

#### mode

is the mode to which STDOUT is to be set. The valid values are:

- taskid** Specifies single mode for STDOUT, where *taskid* is the task identifier of the new single task. This value must be between 0 and *n*-1, where *n* is the total of tasks in the current partition. The *taskid* requested does not have to be the issuing task.
- 2** Specifies ordered mode for STDOUT. The macro `STDIO_ORDERED` is supplied for use in C programs.
- 3** Specifies unordered mode for STDOUT. The macro `STDIO_UNORDERED` is supplied for use in C programs.

### Notes

- All current STDOUT buffers are flushed before the new STDOUT mode is established.
- The initial mode for STDOUT is set by using the environment variable `MP_STDOUTMODE`, or by using the command-line option `-stdoutmode`, with the latter overriding the former. The default STDOUT mode is unordered.
- This subroutine is implemented with a half second sleep interval to ensure that the mode change request is processed before subsequent writes to STDOUT.
- This subroutine is thread safe.

### Return values

In C and C++ calls, the following applies:

- 0** Indicates successful completion.

- 1 Indicates that an error occurred. A message describing the error will be issued.

## Examples

### C Example

The following program uses **poe** with the **-labelio yes** option and three tasks:

```
#include <pm_util.h>

main()
{
    mpc_stdout_mode(STDIO_ORDERED);
    printf("These lines will appear in task order\n");
    /*
     * Call mpc_flush here to make sure that one task
     * doesn't change the mode before all tasks have
     * sent the previous printf string to the home node.
     */
    mpc_flush(1);
    mpc_stdout_mode(STDIO_UNORDERED);
    printf("These lines will appear in the order received by the home node\n");
    /*
     * Since synchronization is not used here, one task could actually
     * execute the next statement before one of the other tasks has
     * executed the previous statement, causing one of the unordered
     * lines not to print.
     */
    mpc_stdout_mode(1);
    printf("Only 1 copy of this line will appear from task 1\n");
}
```

Running the above C program produces the following output (task order of lines 4-6 may differ):

- 0 : These lines will appear in task order.
- 1 : These lines will appear in task order.
- 2 : These lines will appear in task order.
- 1 : These lines will appear in the order received by the home node.
- 2 : These lines will appear in the order received by the home node.
- 0 : These lines will appear in the order received by the home node.
- 1 : Only 1 copy of this line will appear from task 1.

### FORTRAN Example

```
CALL MP_STDOUT_MODE(-2)
WRITE(6, *) 'These lines will appear in task order'
CALL MP_FLUSH(1)
CALL MP_STDOUT_MODE(-3)
WRITE(6, *) 'These lines will appear in the order received by the home node'
CALL MP_STDOUT_MODE(1)
WRITE(6, *) 'Only 1 copy of this line will appear from task 1'
END
```

Running the above program produces the following output (the task order of lines 4 through 6 may differ):

- 0 : These lines will appear in task order.
- 1 : These lines will appear in task order.
- 2 : These lines will appear in task order.
- 1 : These lines will appear in the order received by the home node.
- 2 : These lines will appear in the order received by the home node.
- 0 : These lines will appear in the order received by the home node.

## MP\_STDOUT\_MODE

- 1 : Only 1 copy of this line will appear from task 1.

### Related information

Commands:

- mpcc\_r
- mpCC\_r
- mpxlf\_r

Subroutines:

- MP\_FLUSH, mpc\_flush
- MP\_STDOUTMODE\_QUERY, mpc\_stdoutmode\_query
- MP\_SYNCH, mpc\_synch

---

## MP\_STDOUTMODE\_QUERY, mpc\_stdoutmode\_query

### Purpose

Queries the current STDOUT mode setting.

### Library

libmpi\_r.a

### C synopsis

```
#include <pm_util.h>
int mpc_stdoutmode_query(int *mode);
```

### FORTTRAN synopsis

```
MP_STDOUTMODE_QUERY(INTEGER MODE)
```

### Description

This parallel utility subroutine returns the mode to which STDOUT is currently set.

### Parameters

**mode**

is the address of an integer in which the current STDOUT mode setting will be returned. Possible return values are:

- taskid** Indicates that the current STDOUT mode is single, i.e. output for only task taskid is displayed.
- 2** Indicates that the current STDOUT mode is ordered. The macro `STDIO_ORDERED` is supplied for use in C programs.
- 3** Indicates that the current STDOUT mode is unordered. The macro `STDIO_UNORDERED` is supplied for use in C programs.

### Notes

- Between the time one task issues a mode query request and receives a response, it is possible that another task can change the STDOUT mode setting to another value unless proper synchronization is used.
- This subroutine is thread safe.

### Return values

In C and C++ calls, the following applies:

- 0** Indicates successful completion
- 1** Indicates that an error occurred. A message describing the error will be issued.

### Examples

#### C Example

The following program uses **poe** with one task:

```
#include <pm_util.h>

main()
```

## MP\_STDOUTMODE\_QUERY

```
{
    int mode;

    mpc_stdoutmode_query(&mode);
    printf("Initial (default) STDOUT mode is %d\n", mode);
    mpc_stdout_mode(STDIO_ORDERED);
    mpc_stdoutmode_query(&mode);
    printf("New STDOUT mode is %d\n", mode);
}
```

Running the above program produces the following output:

- Initial (default) STDOUT mode is -3
- New STDOUT mode is -2

### FORTRAN Example

The following program uses **poe** with one task:

```
INTEGER MODE
```

```
CALL MP_STDOUTMODE_QUERY(mode)
WRITE(6, *) 'Initial (default) STDOUT mode is', mode
CALL MP_STDOUT_MODE(-2)
CALL MP_STDOUTMODE_QUERY(mode)
WRITE(6, *) 'New STDOUT mode is', mode
END
```

Running the above program produces the following output:

- Initial (default) STDOUT mode is -3
- New STDOUT mode is -2

## Related information

Commands:

- mpcc\_r
- mpCC\_r
- mpxlf\_r

Subroutines:

- MP\_FLUSH, mpc\_flush
- MP\_STDOUT\_MODE, mpc\_stdout\_mode
- MP\_SYNC, mpc\_synch

---

## MP\_UNSET\_CKPT\_CALLBACKS, mpc\_unset\_ckpt\_callbacks

### Purpose

Unregisters checkpoint, resume, and restart application callbacks.

### Library

libmpi\_r.a

### C synopsis

```
#include <pm_ckpt.h>
int mpc_unset_ckpt_callbacks(int handle);
```

### FORTRAN synopsis

```
MP_UNSET_CKPT_CALLBACKS(INTEGER HANDLE, INTEGER RC)
```

### Description

The MP\_UNSET\_CKPT\_CALLBACKS subroutine is called to unregister checkpoint, resume, and restart application callbacks that were registered with the MP\_SET\_CKPT\_CALLBACKS subroutine.

### Parameters

*handle* is an integer indicating the set of callback subroutines to be unregistered. This integer is the value returned by the subroutine used to register the callback subroutine.

In FORTRAN, *RC* contains one of the values listed under **Return Values**.

### Notes

If a call to MP\_UNSET\_CKPT\_CALLBACKS is issued while a checkpoint is in progress, it is possible that the previously-registered callback will still be run during this checkpoint.

There are certain limitations associated with checkpointing an application. See “Checkpoint and restart limitations” on page 39 for more information.

For general information on checkpointing and restarting programs, see *IBM Parallel Environment for AIX: Operation and Use, Volume 1*.

For more information on the use of LoadLeveler and checkpointing, see *IBM LoadLeveler for AIX 5L: Using and Administering*.

### Return values

- 0 Indicates that MP\_UNSET\_CKPT\_CALLBACKS successfully removed the callback subroutines from the list of registered callback subroutines
- 1 Indicates that an error occurred. A message describing the error will be issued.

### Examples

#### C Example

## MP\_UNSET\_CKPT\_CALLBACKS

```
#include <pm_ckpt.h>
int ihndl;
callbacks_t cbs;
void foo(void);
void bar(void);
cbs.checkpoint_callback=foo;
cbs.resume_callback=bar;
cbs.restart_callback=bar;
ihndl = mpc_set_ckpt_callbacks(callbacks_t *cbs);
:
mpc_unset_ckpt_callbacks(ihndl);
:
```

### FORTRAN Example

```
SUBROUTINE FOO
:
:
RETURN
END
SUBROUTINE BAR
:
:
RETURN
END
PROGRAM MAIN
EXTERNAL FOO, BAR
INTEGER HANDLE, RC
:
CALL MP_SET_CKPT_CALLBACKS(FOO,BAR,BAR,HANDLE)
IF (HANDLE .NE. 0) STOP 666
:
:
CALL MP_UNSET_CKPT_CALLBACKS(HANDLE,RC)
:
:
END
```

## Related information

Commands:

- poeckpt
- poerestart

Subroutines:

- MP\_INIT\_CKPT, mpc\_init\_ckpt
- MP\_SET\_CKPT\_CALLBACKS, mpc\_set\_ckpt\_callbacks

---

## pe\_dbg\_breakpoint

### Purpose

Provides a communication mechanism between Parallel Operating Environment (POE) and an attached third party debugger (TPD).

### Library

POE API library (libpoeapi.a)

### C synopsis

```
#include <pe_dbg_checkpoint.h>
void pe_dbg_breakpoint(void);
```

### Description

The **pe\_dbg\_breakpoint** subroutine is used to exchange information between POE and an attached TPD for the purposes of starting, checkpointing, or restarting a parallel application. The call to the subroutine is made by the POE application within the context of various debug events and related POE global variables, which may be examined or filled in by POE and the TPD. All task-specific arrays are allocated by POE and should be indexed by task number (starting with 0) to retrieve or set information specific to that task.

The TPD should maintain a breakpoint within this function, check the value of **pe\_dbg\_debugevent** when the function is entered, take the appropriate actions for each event as described below, and allow the POE executable to continue.

#### PE\_DBG\_INIT\_ENTRY

Used by POE to determine if a TPD is present. The TPD should set the following:

##### int pe\_dbg\_stoptask

Should be set to 1 if a TPD is present. POE will then cause the remote applications to be stopped using **ptrace**, allowing the TPD to attach to and continue the tasks as appropriate.

In addition, POE will interpret the SIGSOUND and SIGRETRACT signals as checkpoint requests from the TPD. SIGSOUND should be sent when the parallel job should continue after a successful checkpoint, and SIGRETRACT should be sent when the parallel job should terminate after a successful checkpoint.

**Note:** Unpredictable results may occur if these signals are sent while a parallel checkpoint from a **PE\_DBG\_CKPT\_REQUEST** is still in progress.

#### PE\_DBG\_CREATE\_EXIT

Indicates that all remote tasks have been created and are stopped. The TPD may retrieve the following information about the remote tasks:

##### int pe\_dbg\_count

The number of remote tasks that were created. Also the number of elements in task-specific arrays in the originally started process, which remains constant across restarts.

For a restarted POE process, this number may not be the same as the number of tasks that existed when POE was originally started. To

## pe\_dbg\_breakpoint

determine which tasks may have exited prior to the checkpoint from which the restart is performed, the **poe\_task\_info** routine should be used.

**long \*pe\_dbg\_hosts**

Address of the array of remote task host IP addresses.

**long \*pe\_dbg\_pids**

Address of the array of remote task process IDs. Each of these will also be used as the **chk\_pid** field of the **cstate** structure for that task's checkpoint.

**char \*\*pe\_dbg\_executables**

Address of the array of remote task executable names, excluding path.

### PE\_DBG\_CKPT\_REQUEST

Indicates that POE has received a user-initiated checkpoint request from one or all of the remote tasks, has received a request from LoadLeveler to checkpoint an interactive job, or has detected a pending checkpoint while being run as a LoadLeveler batch job. The TPD should set the following:

**int pe\_dbg\_do\_ckpt**

Should be set to 1 if the TPD wishes to proceed with the checkpoint.

### PE\_DBG\_CKPT\_START

Used by POE to inform the TPD whether or not to issue a checkpoint of the POE process. The TPD may retrieve or set the following information for this event:

**int pe\_dbg\_ckpt\_start**

Indicates that the checkpoint may proceed if set to 1, and the TPD may issue a **pe\_dbg\_checkpoint** of the POE process and some or all of the remote tasks.

The TPD should obtain (or derive) the checkpoint file names, checkpoint flags, **cstate**, and checkpoint error file names from the variables below.

**char \*pe\_dbg\_poe\_ckptfile**

Indicates the full pathname to the POE checkpoint file to be used when checkpointing the POE process. The name of the checkpoint error file can be derived from this name by concatenating the **.err** suffix. The checkpoint error file name should also be used for **PE\_DBG\_CKPT\_START** events to know the file name from which to read the error data.

**char \*\*pe\_dbg\_task\_ckptfiles**

Address of the array of full pathnames to be used for each of the task checkpoints. The name of the checkpoint error file can be derived from this name by concatenating the **.err** suffix.

**int pe\_dbg\_poe\_ckptflags**

Indicates the checkpoint flags to be used when checkpointing the POE process. Other supported flag values for terminating or stopping the POE process may be ORed in by the TPD, if the TPD user issued the checkpoint request.

**int pe\_dbg\_task\_ckptflags**

Indicates the checkpoint flags to be used when checkpointing the remote tasks. Other supported flag values for stopping the remote tasks must be ORed in by the TPD.

The **id** argument for calls to the **pe\_dbg\_checkpoint** routine may be derived from the checkpoint flags. If **CHKPNT\_CRID** is set in the checkpoint flags, the **pe\_dbg\_getcrid** routine should be used to determine the CRID of the checkpoint/restart group. Otherwise, the PID of the target process should be used. Note that the **CHKPNT\_CRID** flag will always be set for the remote task checkpoints, and may or may not be set for POE checkpoints.

**int pe\_dbg\_task\_pipecnt**

Indicates the number of pipefds that will appear for each task in the **pe\_dbg\_task\_pipefds** array. This value must also be used for **chk\_nfd** in the **cstate** structure of the remote task checkpoints.

**int \*\*pe\_dbg\_task\_pipefds**

Pointer to the arrays containing the file descriptor numbers for each of the remote tasks. These numbers must be used for **chk\_fdp** in the **cstate** structure of the remote task checkpoints.

The following variable should be examined by the TPD, but contains no information directly related to making the **pe\_dbg\_checkpoint** calls.

**int pe\_dbg\_ckpt\_aware**

Indicates whether or not the remote tasks that make up the parallel application are checkpoint aware.

The following variables should be filled in by the TPD prior to continuing POE from this event:

**int \*pe\_dbg\_ckpt\_pmd**

Address of an array used by the TPD to indicate which tasks will have the checkpoints performed by the TPD (value=0) and which tasks the Partition Manager Daemon (PMD) should issue checkpoints for (value=1). POE requires that the TPD must perform all checkpoints for a particular parallel job on any node where at least one checkpoint will be performed by the TPD.

**int pe\_dbg\_brkpt\_len**

Used to inform POE of how much data to allocate for **pe\_dbg\_brkpt\_data** for later use by the TPD when saving or restoring breakpoint data. A value of 0 may be used when there is no breakpoint data.

**PE\_DBG\_CKPT\_START\_BATCH**

Same as **PE\_DBG\_CKPT\_START**, but the following variables should be ignored:

- **int pe\_dbg\_ckpt\_start**
- **int pe\_dbg\_poe\_ckptflags**

For this event, the TPD should not issue a checkpoint of the POE process.

**PE\_DBG\_CKPT\_VERIFY**

Indicates that POE has detected a pending checkpoint. POE must verify that the checkpoint was issued by the TPD before proceeding. The TPD should set the following:

**int pe\_dbg\_is\_tpd**

Should be set to 1 if the TPD issued the checkpoint request.

**PE\_DBG\_CKPT\_STATUS**

Indicates the status of the remote checkpoints that were performed by the TPDs. The TPD should set the following:

## pe\_dbg\_breakpoint

### int \*pe\_dbg\_task\_ckpterrno

Address of the array of errno values from the remote task checkpoints (0 for successful checkpoint). These values can be obtained from the `Py_error` field of the `cr_error_t` struct, returned from the `pe_dbg_read_cr_errfile` calls.

### void \*pe\_dbg\_brkpt\_data

The breakpoint data to be included as part of POE's checkpoint file. The format of the data is defined by the TPD, and may be retrieved from POE's address space at restart time.

### int \*pe\_dbg\_Sy\_errors

The secondary errors obtained from `pe_dbg_read_cr_errfile`. These values can be obtained from the `Sy_error` field of the `cr_error_t` struct, returned from the `pe_dbg_read_cr_errfile` calls.

### int \*pe\_dbg\_Xtnd\_errors

The extended errors obtained from `pe_dbg_read_cr_errfile`. These values can be obtained from the `Xtnd_error` field of the `cr_error_t` struct, returned from the `pe_dbg_read_cr_errfile` calls.

### int \*pe\_dbg\_error\_lens

The user error data lengths obtained from `pe_dbg_read_cr_errfile`. These values can be obtained from the `error_len` field of the `cr_error_t` struct, returned from the `pe_dbg_read_cr_errfile` calls.

## PE\_DBG\_CKPT\_ERRDATA

Indicates that the TPD has reported one or more task checkpoint failures, and that POE has allocated space in the following array for the TPD to use to fill in the error data.

### char \*\*pe\_dbg\_error\_data

The user error data obtained from `pe_dbg_read_cr_errfile`. These values can be obtained from the `error_data` field of the `cr_error_t` struct, returned from the `pe_dbg_read_cr_errfile` calls.

## PE\_DBG\_CKPT\_DETACH

Used by POE to indicate to the TPD that it should detach from the POE process. After being continued from `pe_dbg_breakpoint` for this event (just prior to the TPD actually detaching), POE will wait until its trace bit is no longer set before instructing the kernel to write its checkpoint file. POE will indicate to the TPD that it is safe to reattach to the POE process by creating the file `/tmp/.poe.PID.reattach`, where *PID* is the process ID of the POE process.

## PE\_DBG\_CKPT\_RESULTS

Indicates the checkpoint results to either POE or the TPD, depending on who issued the checkpoint of POE.

### int pe\_dbg\_ckpt\_rc

If the TPD issued the checkpoint, this variable should be filled in by the TPD and should contain the return code from the call to `pe_dbg_checkpoint`. Otherwise, POE will fill in this value to indicate to the TPD whether the checkpoint succeeded (value=1) or failed (value=0). For failed checkpoints, the TPD may obtain the error information from the POE checkpoint error file.

### int pe\_dbg\_ckpt\_errno

If the TPD issued the checkpoint and the checkpoint failed, this variable should be filled in by the TPD and should contain the `errno` set by AIX upon return from `pe_dbg_checkpoint`.

**PE\_DBG\_CKPT\_RESUME**

When this event occurs, the TPD may continue or terminate the remote tasks (or keep them stopped) after a successful checkpoint. The TPD must not perform the post-checkpoint actions until this event is received, to ensure that POE and LoadLeveler have performed their post-checkpoint synchronization. If the TPD did not issue the checkpoint, the following variable should be examined:

**int pe\_dbg\_ckpt\_action**

POE will fill in this value to indicate to the TPD if the remote tasks should be continued (value=0) or terminated (value=1) after a successful checkpoint.

**PE\_DBG\_CKPT\_CANCEL**

Indicates that POE has received a request to cancel an in-progress checkpoint. The TPD should cause a SIGINT to be sent to the thread that issued the **pe\_dbg\_checkpoint** calls in the remote tasks. If the TPD is non-threaded and performs non-blocking checkpoints, the task checkpoints cannot be cancelled.

**Note:** If the TPD user issues a request to cancel a checkpoint being performed by the TPD, the TPD should send a SIGGRANT to the POE process so that the remote checkpoints being performed by the PMDs can be interrupted. Otherwise, the checkpoint call in the TPD can return while some remote checkpoints are still in progress.

**PE\_DBG\_RESTART\_READY**

Indicates that processes for the remote task restarts have been created and that **pe\_dbg\_restart** calls for the remote tasks may be issued by the TPD. The TPD must perform the restarts of all remote tasks.

The TPD should first retrieve the remote task information specified in the variables described above under **PE\_DBG\_CREATE\_EXIT**. The TPD should then obtain (or derive) the restart file names, the restart flags, rstate, and restart error file names from the variables below. The **id** argument for the **pe\_dbg\_restart** call must be derived from the remote task PID using **pe\_dbg\_getcrd** routine.

**char \*\*pe\_dbg\_task\_rstfiles**

Address of the array of full pathnames to be used for each of the task restarts. The name of the restart error file can be derived from this name by concatenating the **.err** suffix.

**int pe\_dbg\_task\_rstflags**

Indicates the restart flags to be used when restarting the remote tasks. Other supported flag values for stopping the remote tasks may be ORed in by the TPD.

**char \*\*pe\_dbg\_task\_rstate**

Address of the array of strings containing the restart data required for each of the remote tasks. This value may be used as is for the **rst\_buffer** member of the **rstate** structure used in the remote task restarts, or additional data may be appended by the TPD, as described below:

DEBUGGER\_STOP=yes

If this string appears in the task restart data, followed by a newline (**\n**) character and a **\0**, the remote task will send a SIGSTOP signal to itself once all restart actions have been completed in the restart handler. This will likely be used by the TPD when tasks are

## pe\_dbg\_breakpoint

checkpoint-aware, and the TPD wants immediate control of the task after it completes restart initialization.

The `rst_len` member of the `rstate` structures should include a `\0`, whether the TPD appends to the `rst_buffer` or not.

The following variables should be re-examined by the TPD during this event:

### **int pe\_dbg\_ckpt\_aware**

Indicates whether or not the remote tasks that make up the parallel application are checkpoint aware.

### **void \*pe\_dbg\_brkpt\_data**

The breakpoint data that was included as part of POE's checkpoint file. The format of the data is defined by the TPD.

The following variables should be filled in by the TPD prior to continuing POE from this event. This also implies that all remote restarts must have been performed before continuing POE:

### **int \*pe\_dbg\_task\_rsternnos**

Address of the array of `errno`s from the remote task restarts (0 for successful restart). These values can be obtained from the `Py_error` field of the `cr_error_t` struct, returned from the `pe_dbg_read_cr_errfile` calls.

### **int \*pe\_dbg\_Sy\_errors**

The secondary errors obtained from `pe_dbg_read_cr_errfile`. These values can be obtained from the `Sy_error` field of the `cr_error_t` struct, returned from the `pe_dbg_read_cr_errfile` calls.

### **int \*pe\_dbg\_Xtnd\_errors**

The extended errors obtained from `pe_dbg_read_cr_errfile`. These values can be obtained from the `Xtnd_error` field of the `cr_error_t` struct, returned from the `pe_dbg_read_cr_errfile` calls.

### **int \*pe\_dbg\_error\_lens**

The user error data lengths obtained from `pe_dbg_read_cr_errfile`. These values can be obtained from the `error_len` field of the `cr_error_t` struct, returned from the `pe_dbg_read_cr_errfile` calls.

### **PE\_DBG\_RESTART\_ERRDATA**

Indicates that the TPD has reported one or more task restart failures, and that POE has allocated space in the following array for the TPD to use to fill in the error data.

### **char \*\*pe\_dbg\_error\_data**

The user error data obtained from `pe_dbg_read_cr_errfile`. These values can be obtained from the `error_data` field of the `cr_error_t` struct, returned from the `pe_dbg_read_cr_errfile` calls.

## Notes

Use `-I/usr/lpp/ppe.poe/include` to pick up the header file. This flag is an uppercase letter `i`.

Any references to process ID or PID above represent the real process ID, and not the virtual process ID associated with checkpointed/restarted processes.

---

## pe\_dbg\_checkpoint

### Purpose

Checkpoint a process that is under debugger control, or a group of processes.

### Library

POE API library (libpoeapi.a)

### C synopsis

```
#include <pe_dbg_checkpoint.h>
int pe_dbg_checkpoint(path, id, flags, cstate, epath)
char *path;
id_t id;
unsigned int flags;
chk_state_t *cstate;
char *epath;
```

### Description

The **pe\_dbg\_checkpoint** subroutine allows a process to checkpoint a process that is under debugger control, or a set of processes that have the same checkpoint/restart group ID (CRID). The state information of the checkpointed processes is saved in a single file. All information required to restart the processes (other than the executable files, any shared libraries, any explicitly loaded modules and data, if any, passed through the restart system calls) is contained in the checkpoint file.

Processes to be checkpointed will be stopped before the process information is written to the checkpoint file to maintain data integrity. If a process has not registered a checkpoint handler, it will be stopped when a checkpoint request is issued. However, if a process has registered a checkpoint handler, the debugger must allow the checkpoint handler to reach its call to **checkpoint\_commit** for the process to be put into the stopped state.

After all processes have been stopped, the checkpoint file is written with process information one process at a time. After the write has completed successfully, the **pe\_dbg\_checkpoint** subroutine will do one of the following depending on the value of the flags passed:

- Continue the processes.
- Terminate all the checkpointed processes.
- Leave the processes in the stopped state.

If any one of the processes to be checkpointed is a **setuid** or **setgid** program, the **pe\_dbg\_checkpoint** subroutine will fail, unless the caller has superuser privilege. If shared memory is being used within the set of processes being checkpointed, all processes that use the shared memory must belong to the checkpoint/restart group being checkpointed, or the **pe\_dbg\_checkpoint** subroutine will fail, unless the **CHKPNT\_IGNORE\_SHMEM** flag is set.

The **pe\_dbg\_checkpoint** subroutine may be interrupted, in which case, all processes being checkpointed will continue to run and neither a checkpoint file nor an error file will be created.

## Parameters

### path

The path of the checkpoint file to be created. This file will be created read-only with the ownership set to the user ID of the process invoking the **pe\_dbg\_checkpoint** call.

**id** Indicates the process ID of the process to be checkpointed or the checkpoint/restart group ID or CRID of the set of processes to be checkpointed as specified by a value of the **flags** parameter.

### flags

Determines the behavior of the **pe\_dbg\_checkpoint** subroutine and defines the interpretation of the **id** parameter. The **flags** parameter is constructed by logically ORing the following values, which are defined in the **sys/checkpnt.h** file:

#### CHKPNT\_AND\_STOP

Setting this bit causes the checkpointed processes to be put in a stopped state after a successful checkpoint operation. The processes can be continued by sending them SIGCONT. The default is to checkpoint and continue running the processes.

#### CHKPNT\_AND\_STOPTRC

Setting this bit causes any process that is traced to be put in a stopped state after a successful checkpoint operation. The processes can be continued by sending them SIGCONT. The default is to checkpoint and continue running the processes.

#### CHKPNT\_AND\_TERMINATE

Setting this bit causes the checkpointed processes to be terminated on a successful checkpoint operation. The default is to checkpoint and continue running the processes.

#### CHKPNT\_CRID

Specifies that the **id** parameter is the checkpoint/restart group ID or CRID of the set of processes to be checkpointed.

#### CHKPNT\_IGNORE\_SHMEM

Specifies that shared memory should not be checkpointed.

#### CHKPNT\_NODELAY

Specifies that **pe\_dbg\_checkpoint** will not wait for the completion of the checkpoint call. As soon as all the processes to be checkpointed have been identified, and the checkpoint operation started for each of them, the call will return. The kernel will not provide any status on whether the call was successful. The application must examine the checkpoint file to determine if the checkpoint operation succeeded or not. By default, the **pe\_dbg\_checkpoint** subroutine will wait for all the checkpoint data to be completely written to the checkpoint file before returning.

The **CHKPNT\_AND\_TERMINATE** and **CHKPNT\_AND\_STOP** flags are mutually exclusive. Do not specify them at the same time.

### cstate

Pointer to a structure of type **chk\_state\_t**. This parameter is ignored unless the process is the primary checkpoint process for the pending checkpoint operation. The list of file descriptors that need to be inherited at restart time should be specified in the structure.

**epath**

An error file name to log error and debugging data if the checkpoint fails. This field is mandatory and must be provided.

**Notes**

Use **-I/usr/lpp/ppe.poe/include** to pick up the header file. This flag is an uppercase letter i.

Any references to process ID or PID above represent the real process ID, and not the virtual process ID associated with checkpointed or restarted processes.

**Return values**

Upon successful completion, a value of **CHECKPOINT\_OK** is returned.

If the invoking process is included in the set of processes being checkpointed, and the **CHKPNT\_AND\_TERMINATE** flag is set, this call will not return if the checkpoint is successful because the process will be terminated.

If the **pe\_dbg\_checkpoint** call is unsuccessful, **CHECKPOINT\_FAILED** is returned and the **errno** global variable is set to indicate the error.

If a process that successfully checkpointed itself is restarted, it will return from the **pe\_dbg\_checkpoint** call with a value of **RESTART\_OK**.

**Errors**

The **pe\_dbg\_checkpoint** subroutine is unsuccessful when the global variable **errno** contains one of the following values:

**EACCES**

One of the following is true:

- The file exists, but could not be opened successfully in exclusive mode, or write permission is denied on the file, or the file is not a regular file.
- Search permission is denied on a component of the path prefix specified by the path parameter. Access could be denied due to a secure mount.
- The file does not exist, and write permission is denied for the parent directory of the file to be created.

**EAGAIN**

Either the calling process or one or more of the processes to be checkpointed is already involved in another checkpoint or restart operation.

**EINTR**

Indicates that the checkpoint operation was terminated due to receipt of a signal. No checkpoint file will be created. A call to the **pe\_dbg\_checkpoint\_wait** subroutine should be made when this occurs, to ensure that the processes reach a state where subsequent checkpoint operations will not fail unpredictably.

**EINVAL**

Indicates that a NULL **path** or **epath** parameter was passed in, or an invalid set of flags was set, or an invalid id parameter was passed.

**ENOMEM**

Insufficient memory exists to initialize the checkpoint structures.

**ENOSYS**

One of the following is true:

- The caller of the function is not a debugger.
- The process could not be checkpointed because it violated a restriction.

**ENOTSUP**

One of the processes to be checkpointed is a kernel process or has a kernel-only thread.

**EPERM**

Indicates that the process does not have appropriate privileges to checkpoint one or more of the processes.

**ESRCH**

One of the following is true:

- The process whose process ID was passed, or the checkpoint/restart group whose CRID was passed, does not exist.
- The process whose process ID was passed, or the checkpoint/restart group whose CRID was passed, is not checkpointable because there is no process that had the environment variable **CHECKPOINT** set to **yes** at execution time.
- The indicated checkpoint/restart group does not have a primary checkpoint process.

---

## pe\_dbg\_checkpoint\_wait

### Purpose

Waits for a checkpoint, or pending checkpoint file I/O, to complete.

### Library

POE API library (libpoeapi.a)

### C synopsis

```
#include <pe_dbg_checkpoint.h>
int pe_dbg_checkpoint_wait(id, flags, options)
id_t id;
unsigned int flags;
int *options;
```

### Description

The **pe\_dbg\_checkpoint\_wait** subroutine can be used to:

- Wait for a pending checkpoint issued by the calling thread's process to complete.
- Determine whether a pending checkpoint issued by the calling thread's process has completed, when the **CHKPNT\_NODELAY** flag is specified.
- Wait for any checkpoint file I/O that may be in progress during an interrupted checkpoint to complete.

The **pe\_dbg\_checkpoint\_wait** subroutine will return to the caller once any checkpoint file I/O that may be in progress during an interrupted checkpoint has completed. The **pe\_dbg\_checkpoint** routine does not wait for this file I/O to complete when the checkpoint operation is interrupted. Failure to perform this call after an interrupted checkpoint can cause a process or set of processes to be in a state where subsequent checkpoint operations could fail unpredictably.

### Parameters

**id** Indicates the process ID or the checkpoint/restart group ID (CRID) of the processes for which a checkpoint operation was initiated or interrupted, as specified by a value of the flag parameter.

#### flags

Defines the interpretation of the **id** parameter. The **flags** parameter may contain the following value, which is defined in the **sys/checkpnt.h** file:

#### CHKPNT\_CRID

Specifies that the **id** parameter is the checkpoint/restart group ID or CRID of the set of processes for which a checkpoint operation was initiated or interrupted.

#### CHKPNT\_NODELAY

Specifies that **pe\_dbg\_checkpoint\_wait** will not wait for the completion of the checkpoint call. This flag should not be used when waiting for pending checkpoint file I/O to complete.

#### options

This field is reserved for future use and should be set to NULL.

Future implementations of this function may return the checkpoint error code in this field. Until then, the size of the checkpoint error file can be used in most cases to determine whether the checkpoint succeeded or failed. If the size

of the file is 0, the checkpoint succeeded, otherwise the checkpoint failed and checkpoint error file will contain the error codes. If the file does not exist, the checkpoint most likely failed due to an **EPERM** or **ENOENT** on the checkpoint error file pathname.

### Notes

Use **-I/usr/lpp/ppe.poe/include** to pick up the header file. This flag is an uppercase letter **i**.

Any references to process ID or PID above represent the real process ID, and not the virtual process ID associated with checkpointed/restarted processes.

### Return values

Upon successful completion, a value of 0 is returned, indicating that one of the following is true:

- The pending checkpoint completed.
- There was no pending checkpoint.
- The pending file I/O completed.
- There was no pending file I/O.

If the **pe\_dbg\_checkpoint\_wait** call is unsuccessful, -1 is returned and the **errno** global variable is set to indicate the error.

### Errors

The **pe\_dbg\_checkpoint\_wait** subroutine is unsuccessful when the global variable **errno** contains one of the following values:

#### **EINPROGRESS**

Indicates that the pending checkpoint operation has not completed when the **CHKPNT\_NODELAY** flag is specified.

#### **EINTR**

Indicates that the operation was terminated due to receipt of a signal.

#### **EINVAL**

Indicates that an invalid flag was set.

#### **ENOSYS**

The caller of the function is not a debugger.

#### **ESRCH**

The process whose process ID was passed or the checkpoint/restart group whose CRID was passed does not exist.

---

## pe\_dbg\_getcrid

### Purpose

Returns the checkpoint/restart ID.

### Library

POE API library (libpoeapi.a)

### C synopsis

```
crid_t pe_dbg_getcrid(pid)
pid_t pid;
```

### Description

The **pe\_dbg\_getcrid** subroutine returns the checkpoint/restart group ID (CRID) of the process whose process ID was specified in the **pid** parameter, or the CRID of the calling process if a value of -1 was passed.

### Parameters

**pid**      Either the process ID of a process to obtain its CRID, or -1 to request the CRID of the calling process.

### Notes

Any references to process ID or PID above represent the real process ID, and not the virtual process ID associated with checkpointed/restarted processes.

### Return values

If the process belongs to a checkpoint/restart group, a valid CRID is returned. If the process does not belong to any checkpoint/restart group, a value of zero is returned. For any error, a value of -1 is returned and the **errno** global variable is set to indicate the error.

### Errors

The **pe\_dbg\_getcrid** subroutine is unsuccessful when the global variable **errno** contains one of the following values:

<b>ENOSYS</b>	The caller of the function is not a debugger.
<b>ESRCH</b>	There is no process with a process id equal to <b>pid</b> .

## pe\_dbg\_getrtid

### Purpose

Returns real thread ID of a thread in a specified process given its virtual thread ID.

### Library

POE API library (libpoeapi.a)

### C synopsis

```
#include <pe_dbg_checkpnt.h>

tid_t pe_dbg_getrtid(pid, vtid)
pid_t pid;
tid_t vtid;
```

### Description

The **pe\_dbg\_getrtid** subroutine returns the real thread ID of the specified virtual thread in the specified process.

### Parameters

- pid**      The real process ID of the process containing the thread for which the real thread ID is needed
- vtid**     The virtual thread ID of the thread for which the real thread ID is needed.

### Return values

If the calling process is not a debugger, a value of -1 is returned. Otherwise, the **pe\_dbg\_getrtid** call is always successful. If the process does not exist or has exited or is not a restarted process, or if the provided virtual thread ID does not exist in the specified process, the value passed in the **vtid** parameter is returned. Otherwise, the real thread ID of the thread whose virtual thread ID matches the value passed in the **vtid** parameter is returned.

### Errors

The **pe\_dbg\_getrtid** subroutine is unsuccessful if the following is true:

- ENOSYS**      The caller of the function is not a debugger.

---

## pe\_dbg\_getvtid

### Purpose

Returns virtual thread ID of a thread in a specified process given its real thread ID.

### Library

POE API library (libpoeapi.a)

### C synopsis

```
#include <pe_dbg_checkpnt.h>

tid_t pe_dbg_getvtid(pid, rtid)
pid_t pid;
tid_t rtid
```

### Description

The **pe\_dbg\_getvtid** subroutine returns the virtual thread ID of the specified real thread in the specified process.

### Parameters

- pid**     The real process ID of the process containing the thread for which the real thread ID is needed
- rtid**    The real thread ID of the thread for which the virtual thread ID is needed.

### Return values

If the calling process is not a debugger, a value of -1 is returned.

Otherwise, the **pe\_dbg\_getvtid** call is always successful.

If the process does not exist, the process has exited, the process is not a restarted process, or the provided real thread ID does not exist in the specified process, the value passed in the **rtid** parameter is returned.

Otherwise, the virtual thread ID of the thread whose real thread ID matches the value passed in the **rtid** parameter is returned.

### Errors

The **pe\_dbg\_getvtid** subroutine is unsuccessful if the following is true:

- ENOSYS**     The caller of the function is not a debugger.

## pe\_dbg\_read\_cr\_errfile

### Purpose

Opens and reads information from a checkpoint or restart error file.

### Library

POE API library (libpoeapi.a)

### C synopsis

```
#include <pe_dbg_checkpoint.h>
void pe_dbg_read_cr_errfile(char *path, cr_error_t *err_data, int cr_errno)
```

### Description

The **pe\_dbg\_read\_cr\_errfile** subroutine is used to obtain the error information from a failed checkpoint or restart. The information is returned in the `cr_error_t` structure, as defined in `/usr/include/sys/checkpnt.h`.

### Parameters

**path**

The full pathname to the error file to be read.

**err\_data**

Pointer to a `cr_error_t` structure in which the error information will be returned.

**cr\_errno**

The `errno` from the **pe\_dbg\_checkpoint** or **pe\_dbg\_restart** call that failed. This value is used for the `Py_error` field of the returned structure if the file specified by the **path** parameter does not exist, has a size of 0, or cannot be opened.

### Notes

Use **-I/usr/lpp/ppe.poe/include** to pick up the header file. This flag is an uppercase letter **i**.

## pe\_dbg\_restart

### Purpose

Restarts processes from a checkpoint file.

### Library

POE API library (libpoeapi.a)

### C synopsis

```
#include <pe_dbg_checkpoint.h>
int pe_dbg_restart(path, id, flags, rstate, epath)
char *path;
id_t id;
unsigned int flags;
rst_state_t *rstate;
char *epath;
```

### Description

The **pe\_dbg\_restart** subroutine allows a process to restart all the processes whose state information has been saved in the checkpoint file.

All information required to restart these processes (other than the executable files, any shared libraries and explicitly loaded modules) is recreated from the information from the checkpoint file. Then, a new process is created for each process whose state was saved in the checkpoint file. The only exception is the primary checkpoint process, which overlays an existing process specified by the **id** parameter.

When restarting a single process that was checkpointed, the **id** parameter specifies the process ID of the process to be overlaid. When restarting a set of processes, the **id** parameter specifies the checkpoint/restart group ID of the process to be overlaid, and the **flags** parameter must set **RESTART\_OVER\_CRID**. This process must also be the primary checkpoint process of the checkpoint/restart group. The user ID and group IDs of the primary checkpoint process saved in the checkpoint file should match the user ID and group IDs of the process it will overlay.

After all processes have been re-created successfully, the **pe\_dbg\_restart** subroutine will do one of the following, depending on the value of the **flags** passed:

- Continue the processes from the point where each thread was checkpointed.
- Leave the processes in the stopped state.

A primary checkpoint process inherits attributes from the attributes saved in the file, and also from the process it overlays. Other processes in the checkpoint file obtain their attributes only from the checkpoint file, unless they share some attributes with the primary checkpoint process. In this case, the shared attributes are inherited. Although the resource usage of each checkpointed process is saved in the checkpoint file, the resource usage attributes will be zeroed out when it is restarted and the **getrusage** subroutine will return only resource usage after the last restart operation.

Some new state data can be provided to processes, primary or non-primary, at restart time if they have a checkpoint handler. The handler should have passed in a valid **rst** parameter when it called **checkpoint\_commit** at checkpoint time. At restart time, a pointer to an interface buffer can be passed through the **rstate**

parameter in the **pe\_dbg\_restart** subroutine. The data in the buffer will be copied to the address previously specified in the **rst** parameter by the checkpoint handler before the process is restarted. The format of the interface buffer is entirely application dependent.

If any one of the processes to be restarted is a **setuid** or a **setgid** program, the **pe\_dbg\_restart** subroutine will fail, unless the caller has root privilege.

## Parameters

### path

The path of the checkpoint file to use for the restart. Must be a valid checkpoint file created by a **pe\_dbg\_checkpoint** call.

**id** Indicates the process ID or the checkpoint/restart group ID or CRID of the process that is to be overlaid by the primary checkpoint process as identified by the **flags** parameter.

### flags

Determines the behavior of the **pe\_dbg\_restart** subroutine and defines the interpretation of the **id** parameter. The **flags** parameter is constructed by logically ORing one or more of the following values, which are defined in the **sys/checkpnt.h** file:

#### RESTART\_AND\_STOP

Setting this bit will cause the restarted processes to be put in a stopped state after a successful restart operation. They can be continued by sending them SIGCONT. The default is to restart and resume running the processes at the point where each thread in the process was checkpointed.

#### RESTART\_AND\_STOPTRC

Setting this bit will cause any process that was traced at checkpoint time to be put in a stopped state after a successful restart operation. The processes can be continued by sending them SIGCONT. The default is to restart and resume execution of the processes at the point where each thread in the process was checkpointed.

#### RESTART\_IGNORE\_BADSC

Causes the restart operation not to fail if a kernel extension that was present at checkpoint time is not present at restart time. However, if the restarted program uses any system calls in the missing kernel extension, the program will fail when those calls are used.

#### RESTART\_OVER\_CRID

Specifies that the **id** parameter is the checkpoint/restart group ID or CRID of the process over which the primary checkpoint process will be restarted. There are multiple processes to be restarted.

#### RESTART\_PAG\_ALL

Same as **RESTART\_WAITER\_PAG**.

#### RESTART\_WAITER\_PAG

Ensures that DCE credentials are restored in the restarted process.

### rstate

Pointer to a structure of type **rst\_state\_t**.

### epath

Path to error file to log error and debugging data, if restart fails.

## Notes

Use **-I/usr/lpp/ppe.poe/include** to pick up the header file. This flag is an uppercase letter i.

Any references to process ID or PID above represent the real process ID, and not the virtual process ID associated with checkpointed/restarted processes.

## Return values

Upon successful completion, a value of 0 is returned. Otherwise, a value of -1 is returned and the `errno` global variable is set to indicate the error.

## Errors

The **pe\_dbg\_restart** subroutine is unsuccessful when the global variable `errno` contains one of the following values:

### EACCES

One of the following is true:

- The file exists, but could not be opened successfully in exclusive mode, or write permission is denied on the file, or the file is not a regular file.
- Search permission is denied on a component of the path prefix specified by the path parameter. Access could be denied due to a secure mount.
- The file does not exist, and write permission is denied for the parent directory of the file to be created.

### EAGAIN

One of the following is true:

- The user ID has reached the maximum limit of processes that it can have simultaneously, and the invoking process is not privileged.
- Either the calling process or the target process is involved in another checkpoint or restart operation.

### EFAULT

Copying from the interface buffer failed. The **rstate** parameter points to a location that is outside the address space of the process.

### EINVAL

One of the following is true:

- A NULL path was passed in.
- The checkpoint file contains invalid or inconsistent data.
- The target process is a kernel process.
- The restart data length in the **rstate** structure is greater than **MAX\_RESTART\_DATA**.

### ENOMEM

One of the following is true:

- There is insufficient memory to create all the processes in the checkpoint file.
- There is insufficient memory to allocate the restart structures inside the kernel.

### ENOSYS

One of the following is true:

- The caller of the function is not a debugger.

## **pe\_dbg\_restart**

- One or more processes could not be restarted because a restriction was violated.
- File descriptors or user ID or group IDs are mismatched between the primary checkpoint process and overlaid process.
- The calling process is also the target of the **pe\_dbg\_restart** subroutine.

### **EPERM**

One of the following is true:

- The calling process does not have appropriate privileges to target for overlay by a restarted process, one or more of the processes identified by the **id** parameter.
- The calling process does not have appropriate privileges to restart one or more of the processes in the checkpoint file.

### **ESRCH**

Indicates that there is no process with the process ID specified by the **id** parameter, or there is no checkpoint restart group with the specified CRID.

---

## Chapter 13. Parallel task identification API subroutines

PE includes an API that allows an application to retrieve the process IDs of all POE *master processes*, or *home node processes* that are running on the same node. The information that is retrieved can be used for accounting, or to get more detailed information about the tasks that are spawned by these POE processes.

This chapter includes descriptions of the parallel task identification API subroutines that are available for parallel programming:

- “poe\_master\_tasks” on page 146.
- “poe\_task\_info” on page 147.

## poe\_master\_tasks

### Purpose

Retrieves the list of process IDs of POE master processes currently running on this system.

### C synopsis

```
#include "poeapi.h"
int poe_master_tasks(pid_t **poe_master_pids);
```

### Description

An application invoking this subroutine while running on a given node can retrieve the list of process IDs of all POE master processes that are currently running on the same node. This information can be used for accounting purposes or can be passed to the **poe\_task\_info** subroutine to obtain more detailed information about tasks spawned by these POE master processes.

### Parameters

On return, (*\*poe\_master\_pids*) points to the first element of an array of *pid\_t* elements that contains the process IDs of POE master processes. It is the responsibility of the calling program to free this array. This pointer is NULL if no POE master process is running on this system or if there is an error condition.

### Notes

Use **-I/usr/lpp/ppe.poe/include** to pick up the header file.

If you are using the **-bI:libpoeapi.exp** binder option, **-L/usr/lpp/ppe.poe/lib** is required; otherwise, you will need to use: **-llibpoeapi**.

### Return values

- |                |  |
|----------------|--|
| greater than 0 | Indicates the size of the array that ( <i>*poe_master_pids</i> ) points to |
| 0              | Indicates that no POE master process is running.                           |
| -1             | Indicates that a system error has occurred.                                |
| -2             | Indicates that POE is unable to allocate memory.                           |
| -3             | Indicates a non-valid <i>poe_master_pids</i> argument.                     |

### Related information

- poe\_task\_info

---

## poe\_task\_info

### Purpose

Returns a NULL-terminated array of pointers to structures of type POE\_TASKINFO.

### C synopsis

```
#include "poeapi.h"
int poe_task_info(pid_t poe_master_pid, POE_TASKINFO ***poe_taskinfo);
```

### Description

Given the process ID of a POE master process, this subroutine returns to the calling program through the *poe\_taskinfo* argument a NULL-terminated array of pointers to structures of type POE\_TASKINFO. There is one POE\_TASKINFO structure for each POE task spawned by this POE master process on a local or remote node.

Each POE\_TASKINFO structure contains:

- node name
- IP address
- task ID
- AIX session ID
- child process name
- child process ID

### Parameters

#### **poe\_master\_pid**

Specifies the process ID of a POE master process.

#### **poe\_taskinfo**

On return, points to the first element of a NULL-terminated array of pointers to structures of type POE\_TASKINFO.

This pointer is NULL if there is an error condition. It is the responsibility of the calling program to free the array of pointers to POE\_TASKINFO structures, as well as the relevant POE\_TASKINFO structures and the subcomponents **h\_name**, **h\_addr**, and **p\_name**.

The structure POE\_TASKINFO is defined in **poeapi.h**:

```
typedef struct POE_TASKINFO {
    char *h_name; /* host name */
    char *ip_addr; /* IP address */
    int task_id; /* task ID */
    int session_id; /* AIX session ID */
    pid_t pid; /* child process ID */
    char *p_name; /* child process name */
} POE_TASKINFO;
```

### Notes

Use **-I/usr/lpp/ppe.poe/include** to pick up the header file.

If you are using the **-bI:libpoeapi.exp** binder option, **-L/usr/lpp/ppe.poe/lib** is required; otherwise, you will need to use: **-llibpoeapi**.

## Return values

**greater than 0**

Indicates the size of the array that (*\*poe\_taskinfo*) points to

**0** Indicates that no POE master process is running or that task information is not available yet

**-1** Indicates that a system error has occurred.

**-2** Indicates that POE is unable to allocate memory.

**-3** Indicates a non-valid *poe\_master\_pids* argument.

## Related information

- poe\_master\_tasks

## Appendix A. MPE subroutine summary

Table 23 lists the non-blocking collective communication subroutines that are available for parallel programming. These subroutines, which have a prefix of **MPE\_I**, are extensions of the MPI standard. They are part of IBM's implementation of the MPI standard for PE. For descriptions of these subroutines, see *IBM Parallel Environment for AIX: MPI Subroutine Reference*.

With PE Version 4, these nonstandard extensions remain available, but their use is deprecated. The implementation of these routines depends on hidden message passing threads. These routines may **not** be used with environment variable **MP\_SINGLE\_THREAD** set to **yes**.

Earlier versions of PE/MPI allowed matching of blocking (MPI) with non-blocking (MPE\_I) collectives. With PE Version 4, it is advised that you do not match blocking and non-blocking collectives in the same collective operation. If you do, a hang situation can occur. It is possible that some existing applications may hang, when run using PE Version 4. In the case of an unexpected hang, turn on DEVELOP mode by setting the environment variable **MP\_EUIDEVELOP** to **yes**, and rerun your application. DEVELOP mode will detect and report any mismatch. If DEVELOP mode identifies a mismatch, you may continue to use the application as is, by setting **MP\_SHARED\_MEMORY** to **no**. If possible, alter the application to remove the matching of non-blocking with blocking collectives.

Table 23. MPE Subroutines

Subroutine:	
C Name	
FORTTRAN Name	Description
MPE_Iallgather MPE_IALLGATHER	non-blocking allgather operation.
MPE_Iallgatherv MPE_IALLGATHERV	non-blocking allgatherv operation.
MPE_Iallreduce MPE_IALLREDUCE	non-blocking allreduce operation.
MPE_Ialltoall MPE_IALLTOALL	non-blocking alltoall operation.
MPE_Ialltoallv MPE_IALLTOALLV	non-blocking alltoallv operation.
MPE_Ibarrier MPE_IBARRIER	non-blocking barrier operation.
MPE_Ibcast MPE_IBCAST	non-blocking broadcast operation.
MPE_Igather MPE_IGATHER	non-blocking gather operation.
MPE_Igatherv MPE_IGATHERV	non-blocking gatherv operation.
MPE_Ireduce MPE_IREDUCE	non-blocking reduce operation.
MPE_Ireduce_scatter MPE_IREDUCE_SCATTER	non-blocking reduce_scatter operation.

Table 23. MPE Subroutines (continued)

Subroutine:	
C Name	
FORTRAN Name	Description
MPE_Iscan MPE_ISCAN	non-blocking scan operation.
MPE_Iscatter MPE_ISCATTER	non-blocking scatter operation.
MPE_Iscatterv MPE_ISCATTERV	non-blocking scatterv operation.

---

## Appendix B. MPE subroutine bindings

Table 24 summarizes the binding information for all of the MPE subroutines listed in *IBM Parallel Environment for AIX: MPI Subroutine Reference*. With PE Version 4, these nonstandard extensions remain available, but their use is deprecated. The implementation of these routines depends on hidden message passing threads. These routines may **not** be used with environment variable **MP\_SINGLE\_THREAD** set to **yes**.

Earlier versions of PE/MPI allowed matching of blocking (MPI) with non-blocking (MPE\_I) collectives. With PE Version 4, it is advised that you do not match blocking and non-blocking collectives in the same collective operation. If you do, a hang situation can occur. It is possible that some existing applications may hang, when run using PE Version 4. In the case of an unexpected hang, turn on DEVELOP mode by setting the environment variable **MP\_EUIDEVELOP** to **yes**, and rerun your application. DEVELOP mode will detect and report any mismatch. If DEVELOP mode identifies a mismatch, you may continue to use the application as is, by setting **MP\_SHARED\_MEMORY** to **no**. If possible, alter the application to remove the matching of non-blocking with blocking collectives.

**Note:** FORTRAN refers to FORTRAN 77 bindings that are officially supported for MPI. However, FORTRAN 77 bindings can be used by FORTRAN 90. FORTRAN 90 and High Performance FORTRAN (HPF) offer array section and assumed shape arrays as parameters on calls. These are not safe with MPI.

---

### Bindings for non-blocking collective communication

Table 24 lists the C and FORTRAN bindings for non-blocking collective communication subroutines. These subroutines, which have a prefix of **MPE\_I**, are extensions of the MPI standard. They are part of IBM's implementation of the MPI standard for PE.

Table 24. Bindings for non-blocking collective communication

C and FORTRAN subroutine	C and FORTRAN binding
MPE_Iallgather	int MPE_Iallgather(void* sendbuf,int sendcount,MPI_Datatype sendtype,void* recvbuf,int recvcnt,MPI_Datatype recvtpe, MPI_Comm comm,MPI_Request *request);
MPE_IALLGATHER	MPE_IALLGATHER(CHOICE SENDBUF,INTEGER SENDCOUNT,INTEGER SENDTYPE, CHOICE RECVBUF,INTEGER RECVCOUNT,INTEGER RECVTPE,INTEGER COMM,INTEGER REQUEST,INTEGER IERROR)
MPE_Iallgatherv	int MPE_Iallgatherv(void* sendbuf,int sendcount,MPI_Datatype sendtype,void* recvbuf,int *recvcnts,int *displs,MPI_Datatype recvtpe,MPI_Comm comm,MPI_Request *request);
MPE_IALLGATHERV	MPE_IALLGATHERV(CHOICE SENDBUF,INTEGER SENDCOUNT,INTEGER SENDTYPE, CHOICE RECVBUF,INTEGER RECVCOUNTS(*),INTEGER DISPLS(*),INTEGER RECVTPE,INTEGER COMM,INTEGER REQUEST,INTEGER IERROR)
MPE_Iallreduce	int MPE_Iallreduce(void* sendbuf,void* recvbuf,int count,MPI_Datatype datatype,MPI_Op op,MPI_Comm comm,MPI_Request *request);

Table 24. Bindings for non-blocking collective communication (continued)

C and FORTRAN subroutine	C and FORTRAN binding
MPE_IALLREDUCE	MPE_IALLREDUCE(CHOICE SENDBUF,CHOICE RECVBUF,INTEGER COUNT,INTEGER DATATYPE,INTEGER OP,INTEGER COMM,INTEGER REQUEST,INTEGER IERROR)
MPE_Ialltoall	int MPE_Ialltoall(void* sendbuf,int sendcount,MPI_Datatype sendtype,void* recvbuf,int recvcount,MPI_Datatype recvtype,MPI_Comm comm,MPI_Request *request);
MPE_IALLTOALL	MPE_IALLTOALL(CHOICE SENDBUF,INTEGER SENDCOUNT,INTEGER SENDTYPE,CHOICE RECVBUF,INTEGER RECVCOUNT,INTEGER RECVTYPE,INTEGER COMM,INTEGER REQUEST,INTEGER IERROR)
MPE_Ialltoallv	int MPE_Ialltoallv(void* sendbuf,int *sendcounts,int *sdispls,MPI_Datatype sendtype,void* recvbuf,int *recvcounts,int *rdispls,MPI_Datatype recvtype,MPI_Comm comm,MPI_Request *request);
MPE_IALLTOALLV	MPE_IALLTOALLV(CHOICE SENDBUF,INTEGER SENDCOUNTS(*),INTEGER SDISPLS(*),INTEGER SENDTYPE,CHOICE RECVBUF,INTEGER RECVCOUNTS(*),INTEGER RDISPLS(*),INTEGER RECVTYPE,INTEGER COMM,INTEGER REQUEST,INTEGER IERROR)
MPE_Ibarrier	int MPE_Ibarrier(MPI_Comm comm, MPI_Request *request);
MPE_IBARRIER	MPE_IBARRIER(INTEGER COMM, INTEGER REQUEST, INTEGER IERROR)
MPE_Ibcast	int MPE_Ibcast(void* buffer, int count, MPI_Datatype datatype, int root, MPI_Comm comm, MPI_Request *request);
MPE_IBCAST	MPE_IBCAST(CHOICE BUFFER, INTEGER COUNT, INTEGER DATATYPE, INTEGER ROOT, INTEGER COMM, INTEGER REQUEST, INTEGER IERROR)
MPE_Igather	int MPE_Igather(void* sendbuf, int sendcount, MPI_Datatype sendtype, void* recvbuf, int recvcount, MPI_Datatype recvtype, int root, MPI_Comm comm, MPI_Request *request);
MPE_IGATHER	MPE_IGATHER(CHOICE SENDBUF, INTEGER SENDCOUNT, INTEGER SENDTYPE, CHOICE RECVBUF, INTEGER RECVCOUNT, INTEGER RECVTYPE, INTEGER ROOT, INTEGER COMM, INTEGER REQUEST, INTEGER IERROR)
MPE_Igatherv	int MPE_Igatherv(void* sendbuf,int sendcount,MPI_Datatype sendtype,void* recvbuf,int *recvcounts,int *displs,MPI_Datatype recvtype,int root,MPI_Comm comm,MPI_Request *request);
MPE_IGATHERV	MPE_IGATHERV(CHOICE SENDBUF,INTEGER SENDCOUNT,INTEGER SENDTYPE, CHOICE RECVBUF,INTEGER RECVCOUNTS(*),INTEGER DISPLS(*),INTEGER RECVTYPE,INTEGER ROOT,INTEGER COMM,INTEGER REQUEST,INTEGER IERROR)
MPE_Ireduce	int MPE_Ireduce(void* sendbuf,void* recvbuf,int count,MPI_Datatype datatype,MPI_Op op,int root,MPI_Comm comm,MPI_Request *request);
MPE_IREDUCE	MPE_IREDUCE(CHOICE SENDBUF,CHOICE RECVBUF,INTEGER COUNT,INTEGER DATATYPE,INTEGER OP,INTEGER ROOT,INTEGER COMM,INTEGER REQUEST,INTEGER IERROR)
MPE_Ireduce_scatter	int MPE_Ireduce_scatter(void* sendbuf,void* recvbuf,int *recvcounts,MPI_Datatype datatype,MPI_Op op,MPI_Comm comm,MPI_Request *request);
MPE_IREDUCE_SCATTER	MPE_IREDUCE_SCATTER(CHOICE SENDBUF,CHOICE RECVBUF,INTEGER RECVCOUNTS(*),INTEGER DATATYPE,INTEGER OP,INTEGER COMM,INTEGER REQUEST,INTEGER IERROR)

Table 24. Bindings for non-blocking collective communication (continued)

C and FORTRAN subroutine	C and FORTRAN binding
MPE_Iscan	int MPE_Iscan(void* sendbuf,void* recvbuf,int count,MPI_Datatype datatype,MPI_Op op,MPI_Comm comm,MPI_Request *request);
MPE_ISCAN	MPE_ISCAN(CHOICE SENDBUF,CHOICE RECVBUF,INTEGER COUNT,INTEGER DATATYPE,INTEGER OP,INTEGER COMM,INTEGER REQUEST,INTEGER IERROR)
MPE_Iscatter	int MPE_Iscatter(void* sendbuf,int sendcount,MPI_Datatype sendtype,void* recvbuf,int recvcount,MPI_Datatype recvtype,int root,MPI_Comm comm,MPI_Request *request);
MPE_ISCATTER	MPE_ISCATTER(CHOICE SENDBUF,INTEGER SENDCOUNT,INTEGER SENDTYPE,CHOICE RECVBUF,INTEGER RECVCOUNT,INTEGER RECVMODE,INTEGER ROOT,INTEGER COMM,INTEGER REQUEST,INTEGER IERROR)
MPE_Iscatterv	int MPE_Iscatterv(void* sendbuf,int *sendcounts,int *displs,MPI_Datatype sendtype,void* recvbuf,int recvcount,MPI_Datatype recvtype,int root,MPI_Comm comm,MPI_Request *request);
MPE_ISCATTERV	MPE_ISCATTERV(CHOICE SENDBUF,INTEGER SENDCOUNTS(*),INTEGER DISPLS(*),INTEGER SENDTYPE,CHOICE RECVBUF,INTEGER RECVCOUNT,INTEGER RECVMODE,INTEGER ROOT,INTEGER COMM,INTEGER REQUEST,INTEGER IERROR)



---

## Appendix C. MPI subroutine and function summary

Table 25 lists the MPI subroutines and functions that are available for parallel programming. For descriptions of these subroutines and functions, see *IBM Parallel Environment for AIX: MPI Subroutine Reference*.

Table 25. MPI subroutines and functions

Subroutine or function name:		
C		
C++		
FORTRAN	Type	Description
MPI_Abort MPI::Comm::Abort MPI_ABORT	Environment management	Forces all tasks of an MPI job to terminate.
MPI_Accumulate MPI::Win::Accumulate MPI_ACCUMULATE	One-sided communication	Accumulates, according to the specified reduction operation, the contents of the origin buffer to the specified target buffer.
MPI_Add_error_class MPI::Add_error_class MPI_ADD_ERROR_CLASS	External interface	Creates a new error class and returns the value for it.
MPI_Add_error_code MPI::Add_error_code MPI_ADD_ERROR_CODE	External interface	Creates a new error code and returns the value for it.
MPI_Add_error_string MPI::Add_error_string MPI_ADD_ERROR_STRING	External interface	Associates an error string with an error code or class.
MPI_Address (none) MPI_ADDRESS	Derived datatype	Returns the address of a location in memory.
MPI_Allgather MPI::Comm::Allgather MPI_ALLGATHER	Collective communication	Collects messages from each task and distributes the resulting message to each.
MPI_Allgatherv MPI::Comm::Allgatherv MPI_ALLGATHERV	Collective communication	Collects messages from each task and distributes the resulting message to all tasks. Messages can have variable sizes and displacements.
MPI_Alloc_mem MPI::Alloc_mem MPI_ALLOC_MEM	Memory allocation	Allocates storage and returns a pointer to it.
MPI_Allreduce MPI::Comm::Allreduce MPI_ALLREDUCE	Collective communication	Applies a reduction operation.
MPI_Alltoall MPI::Comm::Alltoall MPI_ALLTOALL	Collective communication	Sends a distinct message from each task to every task.
MPI_Alltoallv MPI::Comm::Alltoallv MPI_ALLTOALLV	Collective communication	Sends a distinct message from each task to every task. Messages can have different sizes and displacements.

Table 25. MPI subroutines and functions (continued)

Subroutine or function name:		
C		
C++		
FORTRAN	Type	Description
MPI_Alltoallw MPI::Comm::Alltoallw MPI_ALLTOALLW	Collective communication	Sends a distinct message from each task to every task. Messages can have different datatypes, sizes, and displacements.
MPI_Attr_delete (none) MPI_ATTR_DELETE	Communicator	Removes an attribute value from a communicator.
MPI_Attr_get (none) MPI_ATTR_GET	Communicator	Retrieves an attribute value from a communicator.
MPI_Attr_put (none) MPI_ATTR_PUT	Communicator	Associates an attribute value with a communicator.
MPI_Barrier MPI::Comm::Barrier MPI_BARRIER	Collective communication	Blocks each task until all tasks have called it.
MPI_Bcast MPI::Comm::Bcast MPI_BCAST	Collective communication	Broadcasts a message from <i>root</i> to all tasks in the group.
MPI_Bsend MPI::Comm::Bsend MPI_BSEND	Point-to-point communication	Performs a blocking buffered mode send operation.
MPI_Bsend_init MPI::Comm::Bsend_init MPI_BSEND_INIT	Point-to-point communication	Creates a persistent buffered mode send request.
MPI_Buffer_attach MPI::Attach_buffer MPI_BUFFER_ATTACH	Point-to-point communication	Provides MPI with a message buffer for sending.
MPI_Buffer_detach MPI::Detach_buffer MPI_BUFFER_DETACH	Point-to-point communication	Detaches the current buffer.
MPI_Cancel MPI::Request::Cancel MPI_CANCEL	Point-to-point communication	Marks a non-blocking operation for cancellation.
MPI_Cart_coords MPI::Cartcomm::Get_coords MPI_CART_COORDS	Topology	Translates task rank in a communicator into Cartesian task coordinates.
MPI_Cart_create MPI::Intracomm::Create_cart MPI_CART_CREATE	Topology	Creates a communicator containing topology information.
MPI_Cart_get MPI::Cartcomm::Get_topo MPI_CART_GET	Topology	Retrieves Cartesian topology information from a communicator.
MPI_Cart_map MPI::Cartcomm::Map MPI_CART_MAP	Topology	Computes placement of tasks on the physical processor.

Table 25. MPI subroutines and functions (continued)

Subroutine or function name: C C++ FORTRAN			Type	Description
MPI_Cart_rank MPI::Cartcomm::Get_cart_rank MPI_CART_RANK			Topology	Translates task coordinates into a task rank.
MPI_Cart_shift MPI::Cartcomm::Shift MPI_CART_SHIFT			Topology	Returns shifted source and destination ranks for a task.
MPI_Cart_sub MPI::Cartcomm::Sub MPI_CART_SUB			Topology	Partitions a Cartesian communicator into lower-dimensional subgroups.
MPI_Cartdim_get MPI::Cartcomm::Get_dim MPI_CARTDIM_GET			Topology	Retrieves the number of Cartesian dimensions from a communicator.
MPI_Comm_c2f (none) (none)			Conversion function	Translates a C communicator handle into a FORTRAN handle to the same communicator.
MPI_Comm_call_errhandler MPI::Comm::Call_errhandler MPI_COMM_CALL_ERRHANDLER			External interface	Calls the error handler assigned to the communicator with the error code supplied.
(none) MPI::Comm::Clone (MPI::Cartcomm::Clone, MPI::Graphcomm::Clone, MPI::Intercomm::Clone, MPI::Intracomm::Clone) (none)			Communicator	Creates a new communicator that is a duplicate of an existing communicator.
MPI_Comm_compare MPI::Comm::Compare MPI_COMM_COMPARE			Communicator	Compares the groups and contexts of two communicators.
MPI_Comm_create MPI::Intercomm::Create, MPI::Intracomm::Create MPI_COMM_CREATE			Communicator	Creates a new communicator with a given group.
MPI_Comm_create_errhandler MPI::Comm::Create_errhandler MPI_COMM_CREATE_ERRHANDLER			Communicator	Creates an error handler that can be attached to communicators.
MPI_Comm_create_keyval MPI::Comm::Create_keyval MPI_COMM_CREATE_KEYVAL			Communicator	Generates a new communicator attribute key.
MPI_Comm_delete_attr MPI::Comm::Delete_attr MPI_COMM_DELETE_ATTR			Communicator	Removes an attribute value from a communicator.
MPI_Comm_dup MPI::Cartcomm::Dup, MPI::Graphcomm::Dup, MPI::Intercomm::Dup, MPI::Intracomm::Dup MPI_COMM_DUP			Communicator	Creates a new communicator that is a duplicate of an existing communicator.

Table 25. MPI subroutines and functions (continued)

Subroutine or function name:		
C		
C++		
FORTRAN	Type	Description
MPI_Comm_f2c (none) (none)	Conversion function	Returns a C handle to a communicator.
MPI_Comm_free MPI::Comm::Free MPI_COMM_FREE	Communicator	Marks a communicator for deallocation.
MPI_Comm_free_keyval MPI::Comm::Free_keyval MPI_COMM_FREE_KEYVAL	Communicator	Marks a communicator attribute key for deallocation.
MPI_Comm_get_attr MPI::Comm::Get_attr MPI_COMM_GET_ATTR	Communicator	Retrieves the communicator attribute value identified by the key.
MPI_Comm_get_errhandler MPI::Comm::Get_errhandler MPI_COMM_GET_ERRHANDLER	Communicator	Retrieves the error handler currently associated with a communicator.
MPI_Comm_get_name MPI::Comm::Get_name MPI_COMM_GET_NAME	External interface	Returns the name that was last associated with a communicator.
MPI_Comm_group MPI::Comm::Get_group MPI_COMM_GROUP	Group management	Returns the group handle associated with a communicator.
MPI_Comm_rank MPI::Comm::Get_rank MPI_COMM_RANK	Communicator	Returns the rank of the local task in the group associated with a communicator.
MPI_Comm_remote_group MPI::Intercomm::Get_remote_group MPI_COMM_REMOTE_GROUP	Communicator	Returns the handle of the remote group of an inter-communicator.
MPI_Comm_remote_size MPI::Intercomm::Get_remote_size MPI_COMM_REMOTE_SIZE	Communicator	Returns the size of the remote group of an inter-communicator.
MPI_Comm_set_attr MPI::Comm::Set_attr MPI_COMM_SET_ATTR	Communicator	Attaches the communicator attribute value to the communicator and associates it with the key.
MPI_Comm_set_errhandler MPI::Comm::Set_errhandler MPI_COMM_SET_ERRHANDLER	Communicator	Attaches a new error handler to a communicator.
MPI_Comm_set_name MPI::Comm::Set_name MPI_COMM_SET_NAME	External interface	Associates a name string with a communicator.
MPI_Comm_size MPI::Comm::Get_size MPI_COMM_SIZE	Communicator	Returns the size of the group associated with a communicator.
MPI_Comm_split MPI::Intercomm::Split, MPI::Intracomm::Split MPI_COMM_SPLIT	Communicator	Splits a communicator into multiple communicators based on <i>color</i> and <i>key</i> .
MPI_Comm_test_inter MPI::Comm::Is_inter MPI_COMM_TEST_INTER	Communicator	Returns the type of a communicator (intra-communicator or inter-communicator).

Table 25. MPI subroutines and functions (continued)

Subroutine or function name:		
C		
C++		
FORTRAN	Type	Description
MPI_Dims_create MPI::Compute_dims MPI_DIMS_CREATE	Topology	Defines a Cartesian grid to balance tasks.
MPI_Errhandler_c2f (none) (none)	Conversion function	Translates a C error handler into a FORTRAN handle to the same error handler.
MPI_Errhandler_create (none) MPI_ERRHANDLER_CREATE	Environment management	Registers a user-defined error handler.
MPI_Errhandler_f2c (none) (none)	Conversion function	Returns a C handle to an error handler.
MPI_Errhandler_free MPI::Errhandler::Free MPI_ERRHANDLER_FREE	Environment management	Marks an error handler for deallocation.
MPI_Errhandler_get (none) MPI_ERRHANDLER_GET	Environment management	Gets an error handler associated with a communicator.
MPI_Errhandler_set (none) MPI_ERRHANDLER_SET	Environment management	Associates a new error handler with a communicator.
MPI_Error_class MPI::Get_error_class MPI_ERROR_CLASS	Environment management	Returns the error class for the corresponding error code.
MPI_Error_string MPI::Get_error_string MPI_ERROR_STRING	Environment management	Returns the error string for a given error code.
MPI_Exscan MPI::Intracomm::Exscan MPI_EXSCAN	Collective communication	Performs a prefix reduction on data distributed across the group.
MPI_File_c2f (none) (none)	Conversion function	Translates a C file handle into a FORTRAN handle to the same file.
MPI_File_call_errhandler MPI::File::Call_errhandler MPI_FILE_CALL_ERRHANDLER	External interface	Calls the error handler assigned to the file with the error code supplied.
MPI_File_close MPI::File::Close MPI_FILE_CLOSE	MPI-IO	Closes a file.
MPI_File_create_errhandler MPI::File::Create_errhandler MPI_FILE_CREATE_ERRHANDLER	Environment management	Registers a user-defined error handler that you can associate with an open file.
MPI_File_delete MPI::File::Delete MPI_FILE_DELETE	MPI-IO	Deletes a file after pending operations to the file complete.
MPI_File_f2c (none) (none)	Conversion function	Returns a C handle to a file.

Table 25. MPI subroutines and functions (continued)

Subroutine or function name:		
C		
C++		
FORTRAN	Type	Description
MPI_File_get_amode MPI::File::Get_amode MPI_FILE_GET_AMODE	MPI-IO	Retrieves the access mode specified when the file was opened.
MPI_File_get_atomicity MPI::File::Get_atomicity MPI_FILE_GET_ATOMICITY	MPI-IO	Retrieves the current atomicity mode in which the file is accessed.
MPI_File_get_byte_offset MPI::File::Get_byte_offset MPI_FILE_GET_BYTE_OFFSET	MPI-IO	Allows conversion of an offset.
MPI_File_get_errhandler MPI::File::Get_errhandler MPI_FILE_GET_ERRHANDLER	Environment management	Retrieves the error handler currently associated with a file handle.
MPI_File_get_group MPI::File::Get_group MPI_FILE_GET_GROUP	MPI-IO	Retrieves the group of tasks that opened the file.
MPI_File_get_info MPI::File::Get_info MPI_FILE_GET_INFO	MPI-IO	Returns a new Info object identifying the hints associated with a file.
MPI_File_get_position MPI::File::Get_position MPI_FILE_GET_POSITION	MPI-IO	Returns the current position of the individual file pointer relative to the current file view.
MPI_File_get_position_shared MPI::File::Get_position_shared MPI_FILE_GET_POSITION_SHARED	MPI-IO	Returns the current position of the shared file pointer relative to the current file view.
MPI_File_get_size MPI::File::Get_size MPI_FILE_GET_SIZE	MPI-IO	Retrieves the current file size.
MPI_File_get_type_extent MPI::File::Get_type_extent MPI_FILE_GET_TYPE_EXTENT	MPI-IO	Retrieves the extent of a datatype.
MPI_File_get_view MPI::File::Get_view MPI_FILE_GET_VIEW	MPI-IO	Retrieves the current file view.
MPI_File_iread MPI::File::Iread MPI_FILE_IREAD	MPI-IO	Performs a non-blocking read operation.
MPI_File_iread_at MPI::File::Iread_at MPI_FILE_IREAD_AT	MPI-IO	Performs a non-blocking read operation using an explicit offset.
MPI_File_iread_shared MPI::File::Iread_shared MPI_FILE_IREAD_SHARED	MPI-IO	Performs a non-blocking read operation using the shared file pointer.
MPI_File_iwrite MPI::File::Iwrite MPI_FILE_IWRITE	MPI-IO	Performs a non-blocking write operation.
MPI_File_iwrite_at MPI::File::Iwrite_at MPI_FILE_IWRITE_AT	MPI-IO	Performs a non-blocking write operation using an explicit offset.

Table 25. MPI subroutines and functions (continued)

Subroutine or function name:		
C		
C++		
FORTRAN	Type	Description
MPI_File_irewrite_shared MPI::File::Iwrite_shared MPI_FILE_IWRITE_SHARED	MPI-IO	Performs a non-blocking write operation using the shared file pointer.
MPI_File_open MPI::File::Open MPI_FILE_OPEN	MPI-IO	Opens a file.
MPI_File_preallocate MPI::File::Preallocate MPI_FILE_PREALLOCATE	MPI-IO	Ensures that storage space is allocated for the first <i>size</i> bytes of the file associated with <i>fh</i> .
MPI_File_read MPI::File::Read MPI_FILE_READ	MPI-IO	Reads from a file.
MPI_File_read_all MPI::File::Read_all MPI_FILE_READ_ALL	MPI-IO	Reads from a file collectively.
MPI_File_read_all_begin MPI::File::Read_all_begin MPI_FILE_READ_ALL_BEGIN	MPI-IO	Initiates a split collective read operation from a file.
MPI_File_read_all_end MPI::File::Read_all_end MPI_FILE_READ_ALL_END	MPI-IO	Completes a split collective read operation from a file.
MPI_File_read_at MPI::File::Read_at MPI_FILE_READ_AT	MPI-IO	Reads from a file using an explicit offset.
MPI_File_read_at_all MPI::File::Read_at_all MPI_FILE_READ_AT_ALL	MPI-IO	Reads from a file collectively using an explicit offset.
MPI_File_read_at_all_begin MPI::File::Read_at_all_begin MPI_FILE_READ_AT_ALL_BEGIN	MPI-IO	Initiates a split collective read operation from a file using an explicit offset.
MPI_File_read_at_all_end MPI::File::Read_at_all_end MPI_FILE_READ_AT_ALL_END	MPI-IO	Completes a split collective read operation from a file using an explicit offset.
MPI_File_read_ordered MPI::File::Read_ordered MPI_FILE_READ_ORDERED	MPI-IO	Reads from a file collectively using the shared file pointer.
MPI_File_read_ordered_begin MPI::File::Read_ordered_begin MPI_FILE_READ_ORDERED_BEGIN	MPI-IO	Initiates a split collective read operation from a file using the shared file pointer.
MPI_File_read_ordered_end MPI::File::Read_ordered_end MPI_FILE_READ_ORDERED_END	MPI-IO	Completes a split collective read operation from a file using the shared file pointer.
MPI_File_read_shared MPI::File::Read_shared MPI_FILE_READ_SHARED	MPI-IO	Reads from a file using the shared file pointer.
MPI_File_seek MPI::File::Seek MPI_FILE_SEEK	MPI-IO	Sets a file pointer.

Table 25. MPI subroutines and functions (continued)

Subroutine or function name:		
C		
C++		
FORTRAN	Type	Description
MPI_File_seek_shared MPI::File::Seek_shared MPI_FILE_SEEK_SHARED	MPI-IO	Sets a shared file pointer.
MPI_File_set_atomicsity MPI::File::Set_atomicsity MPI_FILE_SET_ATOMICSITY	MPI-IO	Modifies the current atomicsity mode for an opened file.
MPI_File_set_errhandler MPI::File::Set_errhandler MPI_FILE_SET_ERRHANDLER	Environment management	Associates a new error handler with a file.
MPI_File_set_info MPI::File::Set_info MPI_FILE_SET_INFO	MPI-IO	Specifies new hints for an open file.
MPI_File_set_size MPI::File::Set_size MPI_FILE_SET_SIZE	MPI-IO	Expands or truncates an open file.
MPI_File_set_view MPI::File::Set_view MPI_FILE_SET_VIEW	MPI-IO	Associates a new view with an open file.
MPI_File_sync MPI::File::Sync MPI_FILE_SYNC	MPI-IO	Commits file updates of an open file to storage devices.
MPI_File_write MPI::File::Write MPI_FILE_WRITE	MPI-IO	Writes to a file.
MPI_File_write_all MPI::File::Write_all MPI_FILE_WRITE_ALL	MPI-IO	Writes to a file collectively.
MPI_File_write_all_begin MPI::File::Write_all_begin MPI_FILE_WRITE_ALL_BEGIN	MPI-IO	Initiates a split collective write operation to a file.
MPI_File_write_all_end MPI::File::Write_all_end MPI_FILE_WRITE_ALL_END	MPI-IO	Completes a split collective write operation to a file.
MPI_File_write_at MPI::File::Write_at MPI_FILE_WRITE_AT	MPI-IO	Performs a blocking write operation using an explicit offset.
MPI_File_write_at_all MPI::File::Write_at_all MPI_FILE_WRITE_AT_ALL	MPI-IO	Performs a blocking write operation collectively using an explicit offset.
MPI_File_write_at_all_begin MPI::File::Write_at_all_begin MPI_FILE_WRITE_AT_ALL_BEGIN	MPI-IO	Initiates a split collective write operation to a file using an explicit offset.
MPI_File_write_at_all_end MPI::File::Write_at_all_end MPI_FILE_WRITE_AT_ALL_END	MPI-IO	Completes a split collective write operation to a file using an explicit offset.
MPI_File_write_ordered MPI::File::Write_ordered MPI_FILE_WRITE_ORDERED	MPI-IO	Writes to a file collectively using the shared file pointer.

Table 25. MPI subroutines and functions (continued)

Subroutine or function name: C C++ FORTRAN			Type	Description
MPI_File_write_ordered_begin MPI::File::Write_ordered_begin MPI_FILE_WRITE_ORDERED_BEGIN			MPI-IO	Initiates a split collective write operation to a file using the shared file pointer.
MPI_File_write_ordered_end MPI::File::Write_ordered_end MPI_FILE_WRITE_ORDERED_END			MPI-IO	Completes a split collective write operation to a file using the shared file pointer.
MPI_File_write_shared MPI::File::Write_shared MPI_FILE_WRITE_SHARED			MPI-IO	Writes to a file using the shared file pointer.
MPI_Finalize MPI::Finalize MPI_FINALIZE			Environment management	Terminates all MPI processing.
MPI_Finalized MPI::Is_finalized MPI_FINALIZED			Environment management	Returns <b>true</b> if MPI_FINALIZE has completed.
MPI_Free_mem MPI::Free_mem MPI_FREE_MEM			Memory allocation	Frees a block of storage.
MPI_Gather MPI::Comm::Gather MPI_GATHER			Collective communication	Collects individual messages from each task in a group at the <i>root</i> task.
MPI_Gatherv MPI::Comm::Gatherv MPI_GATHERV			Collective communication	Collects individual messages from each task in <i>comm</i> at the <i>root</i> task. Messages can have different sizes and displacements.
MPI_Get MPI::Win::Get MPI_GET			One-sided communication	Transfers data from a window at the target task to the origin task.
MPI_Get_address MPI::Get_address MPI_GET_ADDRESS			Derived datatype	Returns the address of a location in memory.
MPI_Get_count MPI::Status::Get_count MPI_GET_COUNT			Point-to-point communication	Returns the number of elements in a message.
MPI_Get_elements MPI::Status::Get_elements MPI_GET_ELEMENTS			Derived datatype	Returns the number of basic elements in a message.
MPI_Get_processor_name MPI::Get_processor_name MPI_GET_PROCESSOR_NAME			Environment management	Returns the name of the local processor.
MPI_Get_version MPI::Get_version MPI_GET_VERSION			Environment management	Returns the version of the MPI standard supported.
MPI_Graph_create MPI::Intracomm::Create_graph MPI_GRAPH_CREATE			Topology	Creates a new communicator containing graph topology information.

Table 25. MPI subroutines and functions (continued)

Subroutine or function name:		
C		
C++		
FORTRAN	Type	Description
MPI_Graph_get MPI::Graphcomm::Get_topo MPI_GRAPH_GET	Topology	Retrieves graph topology information from a communicator.
MPI_Graph_map MPI::Graphcomm::Map MPI_GRAPH_MAP	Topology	Computes placement of tasks on the physical processor.
MPI_Graph_neighbors MPI::Graphcomm::Get_neighbors MPI_GRAPH_NEIGHBORS	Topology	Returns the neighbors of the given task.
MPI_Graph_neighbors_count MPI::Graphcomm::Get_neighbors_count MPI_GRAPH_NEIGHBORS_COUNT	Topology	Returns the number of neighbors of the given task.
MPI_Graphdims_get MPI::Graphcomm::Get_dims MPI_GRAPHDIMS_GET	Topology	Retrieves graph topology information from a communicator.
MPI_Grequest_complete MPI::Grequest::Complete MPI_GREQUEST_COMPLETE	External interface	Marks the generalized request complete.
MPI_Grequest_start MPI::Grequest::Start MPI_GREQUEST_START	External interface	Initializes a generalized request.
MPI_Group_c2f (none) (none)	Conversion function	Translates a C group handle into a FORTRAN handle to the same group.
MPI_Group_compare MPI::Group::Compare MPI_GROUP_COMPARE	Group management	Compares the contents of two task groups.
MPI_Group_difference MPI::Group::Difference MPI_GROUP_DIFFERENCE	Group management	Creates a new group that is the difference of two existing groups.
MPI_Group_excl MPI::Group::Excl MPI_GROUP_EXCL	Group management	Removes selected tasks from an existing group to create a new group.
MPI_Group_f2c (none) (none)	Conversion function	Returns a C handle to a group.
MPI_Group_free MPI::Group::Free MPI_GROUP_FREE	Group management	Marks a group for deallocation.
MPI_Group_incl MPI::Group::Incl MPI_GROUP_INCL	Group management	Creates a new group consisting of selected tasks from an existing group.
MPI_Group_intersection MPI::Group::Intersect MPI_GROUP_INTERSECTION	Group management	Creates a new group that is the intersection of two existing groups.
MPI_Group_range_excl MPI::Group::Range_excl MPI_GROUP_RANGE_EXCL	Group management	Creates a new group by excluding selected tasks of an existing group.

Table 25. MPI subroutines and functions (continued)

Subroutine or function name:		
C		
C++		
FORTRAN	Type	Description
MPI_Group_range_incl MPI::Group::Range_incl MPI_GROUP_RANGE_INCL	Group management	Creates a new group consisting of selected ranges of tasks from an existing group.
MPI_Group_rank MPI::Group::Get_rank MPI_GROUP_RANK	Group management	Returns the rank of the local task with respect to group.
MPI_Group_size MPI::Group::Get_size MPI_GROUP_SIZE	Group management	Returns the number of tasks in a group.
MPI_Group_translate_ranks MPI::Group::Translate_ranks MPI_GROUP_TRANSLATE_RANKS	Group management	Converts task ranks of one group into ranks of another group.
MPI_Group_union MPI::Group::Union MPI_GROUP_UNION	Group management	Creates a new group that is the union of two existing groups.
MPI_Ibsend MPI::Comm::Ibsend MPI_IBSEND	Point-to-point communication	Performs a non-blocking buffered send.
MPI_Info_c2f (none) (none)	Conversion function	Translates a C Info object handle into a FORTRAN handle to the same Info object.
MPI_Info_create MPI::Info::Create MPI_INFO_CREATE	Info object	Creates a new, empty Info object.
MPI_Info_delete MPI::Info::Delete MPI_INFO_DELETE	Info object	Deletes a ( <i>key</i> , <i>value</i> ) pair from an Info object.
MPI_Info_dup MPI::Info::Dup MPI_INFO_DUP	Info object	Duplicates an Info object.
MPI_Info_f2c (none) (none)	Conversion function	Returns a C handle to an Info object.
MPI_Info_free MPI::Info::Free MPI_INFO_FREE	Info object	Frees an Info object and sets its handle to MPI_INFO_NULL.
MPI_Info_get MPI::Info::Get MPI_INFO_GET	Info object	Retrieves the value associated with <i>key</i> in an Info object.
MPI_Info_get_nkeys MPI::Info::Get_nkeys MPI_INFO_GET_NKEYS	Info object	Returns the number of keys defined in an Info object.
MPI_Info_get_nthkey MPI::Info::Get_nthkey MPI_INFO_GET_NTHKEY	Info object	Retrieves the <i>n</i> th key defined in an Info object.
MPI_Info_get_valuelen MPI::Info::Get_valuelen MPI_INFO_GET_VALUELEN	Info object	Retrieves the length of the value associated with a key of an Info object.

Table 25. MPI subroutines and functions (continued)

Subroutine or function name:		
C		
C++		
FORTTRAN	Type	Description
MPI_Info_set MPI::Info::Set MPI_INFO_SET	Info object	Adds a ( <i>key, value</i> ) pair to an Info object.
MPI_Init MPI::Init MPI_INIT	Environment management	Initializes MPI.
MPI_Init_thread MPI::Init_thread MPI_INIT_THREAD	Environment management	Initializes MPI and the MPI threads environment.
MPI_Initialized MPI::Is_initialized MPI_INITIALIZED	Environment management	Determines if MPI is initialized.
MPI_Intercomm_create MPI::Intracomm::Create_intercomm MPI_INTERCOMM_CREATE	Communicator	Creates an inter-communicator from two intra-communicators.
MPI_Intercomm_merge MPI::Intercomm::Merge MPI_INTERCOMM_MERGE	Communicator	Creates an intra-communicator by merging the local and remote groups of an inter-communicator.
MPI_Iprobe MPI::Comm::Iprobe MPI_IPROBE	Point-to-point communication	Checks to see if a message matching <i>source, tag, and comm</i> has arrived.
MPI_Irecv MPI::Comm::Irecv MPI_IRECV	Point-to-point communication	Performs a non-blocking receive operation.
MPI_Irsend MPI::Comm::Irsend MPI_IRSEND	Point-to-point communication	Performs a non-blocking ready send operation.
MPI_Is_thread_main MPI::Is_thread_main MPI_IS_THREAD_MAIN	Environment management	Determines whether the calling thread is the thread that called MPI_INIT or MPI_INIT_THREAD.
MPI_Isend MPI::Comm::Isend MPI_ISEND	Point-to-point communication	Performs a non-blocking standard mode send operation.
MPI_Issend MPI::Comm::Issend MPI_ISSEND	Point-to-point communication	Performs a non-blocking synchronous mode send operation.
MPI_Keyval_create (none) MPI_KEYVAL_CREATE	Communicator	Generates a new communicator attribute key.
MPI_Keyval_free (none) MPI_KEYVAL_FREE	Communicator	Marks a communicator attribute key for deallocation.
MPI_Op_c2f (none) (none)	Conversion function	Translates a C reduction operation handle into a FORTRAN handle to the same operation.
MPI_Op_create MPI::Op::Init MPI_OP_CREATE	Collective communication	Binds a user-defined reduction operation to an <i>op</i> handle.

Table 25. MPI subroutines and functions (continued)

Subroutine or function name:		
C		
C++		
FORTRAN	Type	Description
MPI_Op_f2c (none) (none)	Conversion function	Returns a C reduction operation handle to an operation.
MPI_Op_free MPI::Op::Free MPI_OP_FREE	Collective communication	Marks a user-defined reduction operation for deallocation.
MPI_Pack MPI::Datatype::Pack MPI_PACK	Derived datatype	Packs the message in the specified send buffer into the specified buffer space.
MPI_Pack_external MPI::Datatype::Pack_external MPI_PACK_EXTERNAL	Derived datatype	Packs the message in the specified send buffer into the specified buffer space, using the external32 data format.
MPI_Pack_external_size MPI::Datatype::Pack_external_size MPI_PACK_EXTERNAL_SIZE	Derived datatype	Returns the number of bytes required to hold the data, using the external32 data format.
MPI_Pack_size MPI::Datatype::Pack_size MPI_PACK_SIZE	Derived datatype	Returns the number of bytes required to hold the data.
MPI_Pcontrol MPI::Pcontrol MPI_PCONTROL	Environment management	Provides profile control.
MPI_Probe MPI::Comm::Probe MPI_PROBE	Point-to-point communication	Waits until a message matching <i>source</i> , <i>tag</i> , and <i>comm</i> arrives.
MPI_Put MPI::Win::Put MPI_PUT	One-sided communication	Transfers data from the origin task to a window at the target task.
MPI_Query_thread MPI::Query_thread MPI_QUERY_THREAD	Environment management	Returns the current level of threads support.
MPI_Recv MPI::Comm::Recv MPI_RECV	Point-to-point communication	Performs a blocking receive operation.
MPI_Recv_init MPI::Comm::Recv_init MPI_RECV_INIT	Point-to-point communication	Creates a persistent receive request.
MPI_Reduce MPI::Comm::Reduce MPI_REDUCE	Collective communication	Applies a reduction operation to the vector <i>sendbuf</i> over the set of tasks specified by <i>comm</i> and places the result in <i>recvbuf</i> on <i>root</i> .
MPI_Reduce_scatter MPI::Comm::Reduce_scatter MPI_REDUCE_SCATTER	Collective communication	Applies a reduction operation to the vector <i>sendbuf</i> over the set of tasks specified by <i>comm</i> and scatters the result according to the values in <i>recvcounts</i> .

Table 25. MPI subroutines and functions (continued)

Subroutine or function name:		
C		
C++		
FORTRAN	Type	Description
MPI_Register_datarep MPI::Register_datarep MPI_REGISTER_DATAREP	MPI-IO	Registers a data representation.
MPI_Request_c2f (none) (none)	Conversion function	Translates a C request handle into a FORTRAN handle to the same request.
MPI_Request_f2c (none) (none)	Conversion function	Returns a C handle to a request.
MPI_Request_free MPI::Request::Free MPI_REQUEST_FREE	Point-to-point communication	Marks a request for deallocation.
MPI_Request_get_status MPI::Request::Get_status MPI_REQUEST_GET_STATUS	MPI_STATUS object	Accesses the information associated with a request, without freeing the request.
MPI_Rsend MPI::Comm::Rsend MPI_RSEND	Point-to-point communication	Performs a blocking ready mode send operation.
MPI_Rsend_init MPI::Comm::Rsend_init MPI_RSEND_INIT	Point-to-point communication	Creates a persistent ready mode send request.
MPI_Scan MPI::Intracomm::Scan MPI_SCAN	Collective communication	Performs a parallel prefix reduction on data distributed across a group.
MPI_Scatter MPI::Comm::Scatter MPI_SCATTER	Collective communication	Distributes individual messages from <i>root</i> to each task in <i>comm</i> .
MPI_Scatterv MPI::Comm::Scatterv MPI_SCATTERV	Collective communication	Distributes individual messages from <i>root</i> to each task in <i>comm</i> . Messages can have different sizes and displacements.
MPI_Send MPI::Comm::Send MPI_SEND	Point-to-point communication	Blocking standard mode send.
MPI_Send_init MPI::Comm::Send_init MPI_SEND_INIT	Point-to-point communication	Creates a persistent standard mode send request.
MPI_Sendrecv MPI::Comm::Sendrecv MPI_SENDRECV	Point-to-point communication	Performs a blocking send and receive operation.
MPI_Sendrecv_replace MPI::Comm::Sendrecv_replace MPI_SENDRECV_REPLACE	Point-to-point communication	Performs a blocking send and receive operation using a common buffer.
(none) (none) MPI_SIZEOF	Derived datatype	Returns the size in bytes of the machine representation of the given variable.

Table 25. MPI subroutines and functions (continued)

Subroutine or function name:		
C		
C++		
FORTRAN	Type	Description
MPI_Ssend MPI::Comm::Ssend MPI_SSEND	Point-to-point communication	Performs a blocking synchronous mode send operation.
MPI_Ssend_init MPI::Comm::Ssend_init MPI_SSEND_INIT	Point-to-point communication	Creates a persistent synchronous mode send request.
MPI_Start MPI::Prequest::Start MPI_START	Point-to-point communication	Activates a persistent request operation.
MPI_Startall MPI::Prequest::Startall MPI_STARTALL	Point-to-point communication	Activates a collection of persistent request operations.
MPI_Status_c2f (none) (none)	Conversion function	Translates a C status object into a FORTRAN status object.
MPI_Status_f2c (none) (none)	Conversion function	Converts a FORTRAN status object into a C status object.
MPI_Status_set_cancelled MPI::Status::Set_cancelled MPI_STATUS_SET_CANCELLED	External interface	Defines cancellation information for a request.
MPI_Status_set_elements MPI::Status::Set_elements MPI_STATUS_SET_ELEMENTS	External interface	Defines element information for a request.
MPI_Test MPI::Request::Test MPI_TEST	Point-to-point communication	Checks to see if a non-blocking operation has completed.
MPI_Test_cancelled MPI::Status::Is_cancelled MPI_TEST_CANCELLED	Point-to-point communication	Tests whether a non-blocking operation was cancelled.
MPI_Testall MPI::Request::Testall MPI_TESTALL	Point-to-point communication	Tests a collection of non-blocking operations for completion.
MPI_Testany MPI::Request::Testany MPI_TESTANY	Point-to-point communication	Tests for the completion of any specified non-blocking operation.
MPI_Testsome MPI::Request::Testsome MPI_TESTSOME	Point-to-point communication	Tests a collection of non-blocking operations for completion.
MPI_Topo_test MPI::Comm::Get_topology MPI_TOPO_TEST	Topology	Returns the type of virtual topology associated with a communicator.
MPI_Type_c2f (none) (none)	Conversion function	Translates a C datatype handle into a FORTRAN handle to the same datatype.
MPI_Type_commit MPI::Datatype::Commit MPI_TYPE_COMMIT	Derived datatype	Makes a datatype ready for use in communication.

Table 25. MPI subroutines and functions (continued)

Subroutine or function name:		
C		
C++		
FORTRAN	Type	Description
MPI_Type_contiguous MPI::Datatype::Create_contiguous MPI_TYPE_CONTIGUOUS	Derived datatype	Returns a new datatype that represents the concatenation of <i>count</i> instances of <i>oldtype</i> .
MPI_Type_create_darray MPI::Datatype::Create_darray MPI_TYPE_CREATE_DARRAY	Derived datatype	Generates the datatypes corresponding to an HPF-like distribution of an <i>ndims</i> -dimensional array of <i>oldtype</i> elements onto an <i>ndims</i> -dimensional grid of logical tasks.
MPI_Type_create_f90_complex MPI::Datatype::Create_f90_complex MPI_TYPE_CREATE_F90_COMPLEX	Derived datatype	Returns a predefined MPI datatype that matches a COMPLEX variable of KIND selected_real_kind( <i>p</i> , <i>r</i> ).
MPI_Type_create_f90_integer MPI::Datatype::Create_f90_integer MPI_TYPE_CREATE_F90_INTEGER	Derived datatype	Returns a predefined MPI datatype that matches an INTEGER variable of KIND selected_integer_kind( <i>r</i> ).
MPI_Type_create_f90_real MPI::Datatype::Create_f90_real MPI_TYPE_CREATE_F90_REAL	Derived datatype	Returns a predefined MPI datatype that matches a REAL variable of KIND selected_real_kind( <i>p</i> , <i>r</i> ).
MPI_Type_create_hindexed MPI::Datatype::Create_hindexed MPI_TYPE_CREATE_HINDEXED	Derived datatype	Returns a new datatype that represents <i>count</i> blocks. Each block is defined by an entry in <i>array_of_blocklengths</i> and <i>array_of_displacements</i> . Displacements are expressed in bytes.
MPI_Type_create_hvector MPI::Datatype::Create_hvector MPI_TYPE_CREATE_HVECTOR	Derived datatype	Returns a new datatype that represents equally-spaced blocks. The spacing between the start of each block is given in bytes.
MPI_Type_create_indexed_block MPI::Datatype::Create_indexed_block MPI_TYPE_CREATE_INDEXED_BLOCK	Derived datatype	Returns a new datatype that represents <i>count</i> blocks.
MPI_Type_create_keyval MPI::Datatype::Create_keyval MPI_TYPE_CREATE_KEYVAL	Derived datatype	Generates a new attribute key for a datatype.
MPI_Type_create_resized MPI::Datatype::Create_resized MPI_TYPE_CREATE_RESIZED	Derived datatype	Duplicates a datatype and changes the upper bound, lower bound, and extent.
MPI_Type_create_struct MPI::Datatype::Create_struct MPI_TYPE_CREATE_STRUCT	Derived datatype	Returns a new datatype that represents <i>count</i> blocks. Each block is defined by an entry in <i>array_of_blocklengths</i> , <i>array_of_displacements</i> , and <i>array_of_types</i> . Displacements are expressed in bytes.
MPI_Type_create_subarray MPI::Datatype::Create_subarray MPI_TYPE_CREATE_SUBARRAY	Derived datatype	Returns a new datatype that represents an <i>ndims</i> -dimensional subarray of an <i>ndims</i> -dimensional array.

Table 25. MPI subroutines and functions (continued)

Subroutine or function name:		
C		
C++		
FORTRAN	Type	Description
MPI_Type_delete_attr MPI::Datatype::Delete_attr MPI_TYPE_DELETE_ATTR	Derived datatype	Deletes an attribute from a datatype.
MPI_Type_dup MPI::Datatype::Dup MPI_TYPE_DUP	Derived datatype	Duplicates the existing type with associated key values.
MPI_Type_extent (none) MPI_TYPE_EXTENT	Derived datatype	Returns the extent of any defined datatype.
MPI_Type_f2c (none) (none)	Conversion function	Returns a C handle to a datatype.
MPI_Type_free MPI::Datatype::Free MPI_TYPE_FREE	Derived datatype	Marks a derived datatype for deallocation and sets its handle to MPI_DATATYPE_NULL.
MPI_Type_free_keyval MPI::Datatype::Free_keyval MPI_TYPE_FREE_KEYVAL	Derived datatype	Frees a datatype key value.
MPI_Type_get_attr MPI::Datatype::Get_attr MPI_TYPE_GET_ATTR	Derived datatype	Attaches an attribute to a datatype.
MPI_Type_get_contents MPI::Datatype::Get_contents MPI_TYPE_GET_CONTENTS	Derived datatype	Obtains the arguments used in the creation of the datatype.
MPI_Type_get_envelope MPI::Datatype::Get_envelope MPI_TYPE_GET_ENVELOPE	Derived datatype	Determines the constructor that was used to create the datatype.
MPI_Type_get_extent MPI::Datatype::Get_extent MPI_TYPE_GET_EXTENT	Derived datatype	Returns the lower bound and the extent of any defined datatype.
MPI_Type_get_name MPI::Datatype::Get_name MPI_TYPE_GET_NAME	External interface	Returns the name that was last associated with a datatype.
MPI_Type_get_true_extent MPI::Datatype::Get_true_extent MPI_TYPE_GET_TRUE_EXTENT	Derived datatype	Returns the true extent of any defined datatype.
MPI_Type_hindexed (none) MPI_TYPE_HINDEXED	Derived datatype	Returns a new datatype that represents <i>count</i> distinct blocks with offsets expressed in bytes.
MPI_Type_hvector (none) MPI_TYPE_HVECTOR	Derived datatype	Returns a new datatype of <i>count</i> blocks with <i>stride</i> expressed in bytes.
MPI_Type_indexed MPI::Datatype::Create_indexed MPI_TYPE_INDEXED	Derived datatype	Returns a new datatype that represents <i>count</i> blocks with stride in terms of defining type.
MPI_Type_lb (none) MPI_TYPE_LB	Derived datatype	Returns the lower bound of a datatype.

Table 25. MPI subroutines and functions (continued)

Subroutine or function name:		
C		
C++		
FORTRAN	Type	Description
MPI_Type_match_size MPI::Datatype::Match_size MPI_TYPE_CREATE_MATCH_SIZE	Derived datatype	Returns a reference (handle) to one of the predefined named datatypes, not a duplicate.
MPI_Type_set_attr MPI::Datatype::Set_attr MPI_TYPE_SET_ATTR	Derived datatype	Attaches the datatype attribute value to the datatype and associates it with the key.
MPI_Type_set_name MPI::Datatype::Set_name MPI_TYPE_SET_NAME	External interface	Associates a name string with a datatype.
MPI_Type_size MPI::Datatype::Get_size MPI_TYPE_SIZE	Derived datatype	Returns the number of bytes represented by any defined datatype.
MPI_Type_struct (none) MPI_TYPE_STRUCT	Derived datatype	Returns a new datatype that represents <i>count</i> blocks, each with a distinct format and offset.
MPI_Type_ub (none) MPI_TYPE_UB	Derived datatype	Returns the upper bound of a datatype.
MPI_Type_vector MPI::Datatype::Create_vector MPI_TYPE_VECTOR	Derived datatype	Returns a new datatype that represents equally-spaced blocks of replicated data.
MPI_Unpack MPI::Datatype::Unpack MPI_UNPACK	Derived datatype	Unpacks the message into the specified receive buffer from the specified packed buffer.
MPI_Unpack_external MPI::Datatype::Unpack_external MPI_UNPACK_EXTERNAL	Derived datatype	Unpacks the message into the specified receive buffer from the specified packed buffer, using the external32 data format.
MPI_Wait MPI::Request::Wait MPI_WAIT	Point-to-point communication	Waits for a non-blocking operation to complete.
MPI_Waitall MPI::Request::Waitall MPI_WAITALL	Point-to-point communication	Waits for a collection of non-blocking operations to complete.
MPI_Waitany MPI::Request::Waitany MPI_WAITANY	Point-to-point communication	Waits for any specified non-blocking operation to complete.
MPI_Waitsome MPI::Request::Waitsome MPI_WAITSOME	Point-to-point communication	Waits for at least one of a list of non-blocking operations to complete.
MPI_Win_c2f (none) (none)	Conversion function	Translates a C window handle into a FORTRAN handle to the same window.
MPI_Win_call_errhandler MPI::Win::Call_errhandler MPI_WIN_CALL_ERRHANDLER	External interface	Calls the error handler assigned to the window with the error code supplied.

Table 25. MPI subroutines and functions (continued)

Subroutine or function name: C C++ FORTRAN			Type	Description
MPI_Win_complete MPI::Win::Complete MPI_WIN_COMPLETE			One-sided communication	Completes an RMA access epoch on a window object.
MPI_Win_create MPI::Win::Create MPI_WIN_CREATE			One-sided communication	Allows each task in an intra-communicator group to specify a “window” in its memory that is made accessible to accesses by remote tasks.
MPI_Win_create_errhandler MPI::Win::Create_errhandler MPI_WIN_CREATE_ERRHANDLER			One-sided communication	Creates an error handler that can be attached to windows.
MPI_Win_create_keyval MPI::Win::Create_keyval MPI_WIN_CREATE_KEYVAL			One-sided communication	Generates a new window attribute key.
MPI_Win_delete_attr MPI::Win::Delete_attr MPI_WIN_DELETE_ATTR			One-sided communication	Deletes an attribute from a window.
MPI_Win_f2c (none) (none)			Conversion function	Returns a C handle to a window.
MPI_Win_fence MPI::Win::Fence MPI_WIN_FENCE			One-sided communication	Synchronizes RMA calls on a window.
MPI_Win_free MPI::Win::Free MPI_WIN_FREE			One-sided communication	Frees the window object and returns a null handle (equal to MPI_WIN_NULL).
MPI_Win_free_keyval MPI::Win::Free_keyval MPI_WIN_FREE_KEYVAL			One-sided communication	Marks a window attribute key for deallocation.
MPI_Win_get_attr MPI::Win::Get_attr MPI_WIN_GET_ATTR			One-sided communication	Retrieves the window attribute value identified by the key.
MPI_Win_get_errhandler MPI::Win::Get_errhandler MPI_WIN_GET_ERRHANDLER			One-sided communication	Retrieves the error handler currently associated with a window.
MPI_Win_get_group MPI::Win::Get_group MPI_WIN_GET_GROUP			One-sided communication	Returns a duplicate of the group of the communicator used to create a window.
MPI_Win_get_name MPI::Win::Get_name MPI_WIN_GET_NAME			External interface	Returns the name that was last associated with a window.
MPI_Win_lock MPI::Win::Lock MPI_WIN_LOCK			One-sided communication	Starts an RMA access epoch at the target task.
MPI_Win_post MPI::Win::Post MPI_WIN_POST			One-sided communication	Starts an RMA exposure epoch for a local window.

Table 25. MPI subroutines and functions (continued)

Subroutine or function name:		
C		
C++		
FORTRAN	Type	Description
MPI_Win_set_attr MPI::Win::Set_attr MPI_WIN_SET_ATTR	One-sided communication	Attaches the window attribute value to the window and associates it with the key.
MPI_Win_set_errhandler MPI::Win::Set_errhandler MPI_WIN_SET_ERRHANDLER	One-sided communication	Attaches a new error handler to a window.
MPI_Win_set_name MPI::Win::Set_name MPI_WIN_SET_NAME	External interface	Associates a name string with a window.
MPI_Win_start MPI::Win::Start MPI_WIN_START	One-sided communication	Starts an RMA access epoch for a window object.
MPI_Win_test MPI::Win::Test MPI_WIN_TEST	One-sided communication	Tries to complete an RMA exposure epoch.
MPI_Win_unlock MPI::Win::Unlock MPI_WIN_UNLOCK	One-sided communication	Completes an RMA access epoch at the target task.
MPI_Win_wait MPI::Win::Wait MPI_WIN_WAIT	One-sided communication	Completes an RMA exposure epoch.
MPI_Wtick MPI::Wtick MPI_WTICK	Environment management	Returns the resolution of MPI_WTIME in seconds.
MPI_Wtime MPI::Wtime MPI_WTIME	Environment management	Returns the current value of <i>time</i> as a floating-point value.

---

## Appendix D. MPI subroutine bindings

The tables in this appendix summarize the binding information for all of the MPI subroutines listed in *IBM Parallel Environment for AIX: MPI Subroutine Reference*.

**Note:** FORTRAN refers to FORTRAN 77 bindings that are officially supported for MPI. However, FORTRAN 77 bindings can be used by FORTRAN 90. FORTRAN 90 and High Performance FORTRAN (HPF) offer array section and assumed shape arrays as parameters on calls. These are not safe with MPI.

The binding information is divided into these categories:

- “Bindings for collective communication”
- “Bindings for communicators” on page 179
- “Bindings for conversion functions” on page 182
- “Bindings for derived datatypes” on page 183
- “Bindings for environment management” on page 189
- “Bindings for external interfaces” on page 191
- “Bindings for group management” on page 193
- “Bindings for Info objects” on page 195
- “Bindings for memory allocation” on page 196
- “Bindings for MPI-IO” on page 197
- “Bindings for MPI\_Status objects” on page 204
- “Bindings for one-sided communication” on page 205
- “Bindings for point-to-point communication” on page 208
- “Binding for profiling control” on page 213
- “Bindings for topologies” on page 214

---

### Bindings for collective communication

Table 26 lists the bindings for collective communication subroutines.

Table 26. Bindings for collective communication

Subroutine name:	Binding:
C	C
C++	C++
FORTRAN	FORTRAN
MPI_Allgather	int MPI_Allgather(void* sendbuf,int sendcount,MPI_Datatype sendtype,void* recvbuf,int recvcount,MPI_Datatype recvtype, MPI_Comm comm);
MPI::Comm::Allgather	void MPI::Comm::Allgather(const void* sendbuf, int sendcount, const MPI::Datatype& sendtype, void* recvbuf, int recvcount, const MPI::Datatype& recvtype) const;
MPI_ALLGATHER	MPI_ALLGATHER(CHOICE SENDBUF,INTEGER SENDCOUNT,INTEGER SENDTYPE,CHOICE RECVBUFF,INTEGER RECVCOUNT,INTEGER RECVTYP,INTEGER COMM,INTEGER IERROR)

Table 26. Bindings for collective communication (continued)

Subroutine name: C C++ FORTRAN	Binding: C C++ FORTRAN
MPI_Allgatherv	<code>int MPI_Allgatherv(void* sendbuf,int sendcount,MPI_Datatype sendtype,void* recvbuf,int *recvcounts,int *displs, MPI_Datatype recvtype,MPI_Comm comm);</code>
MPI::Comm::Allgatherv	<code>void MPI::Comm::Allgatherv(const void* sendbuf, int sendcount, const MPI::Datatype&amp; sendtype, void* recvbuf, const int recvcounts[], const int displs[], const MPI::Datatype&amp; recvtype) const;</code>
MPI_ALLGATHERV	<code>MPI_ALLGATHERV(CHOICE SENDBUF,INTEGER SENDCOUNT,INTEGER SENDTYPE,CHOICE RECVBUF,INTEGER RECVCOUNTS(*),INTEGER DISPLS(*),INTEGER RECVTYPE,INTEGER COMM,INTEGER IERROR)</code>
MPI_Allreduce	<code>int MPI_Allreduce(void* sendbuf,void* recvbuf,int count,MPI_Datatype datatype,MPI_Op op,MPI_Comm comm);</code>
MPI::Comm::Allreduce	<code>void MPI::Comm::Allreduce(const void* sendbuf, void* recvbuf, int count, const MPI::Datatype&amp; datatype, const MPI::Op&amp; op) const;</code>
MPI_ALLREDUCE	<code>MPI_ALLREDUCE(CHOICE SENDBUF,CHOICE RECVBUF,INTEGER COUNT,INTEGER DATATYPE,INTEGER OP,INTEGER COMM,INTEGER IERROR)</code>
MPI_Alltoall	<code>int MPI_Alltoall(void* sendbuf,int sendcount,MPI_Datatype sendtype,void* recvbuf,int recvcount,MPI_Datatype recvtype, MPI_Comm comm);</code>
MPI::Comm::Alltoall	<code>void MPI::Comm::Alltoall(const void* sendbuf, int sendcount, const MPI::Datatype&amp; sendtype, void* recvbuf, int recvcount, const MPI::Datatype&amp; recvtype) const;</code>
MPI_ALLTOALL	<code>MPI_ALLTOALL(CHOICE SENDBUF,INTEGER SENDCOUNT,INTEGER SENDTYPE,CHOICE RECVBUF,INTEGER RECVCOUNT,INTEGER RECVTYPE,INTEGER COMM,INTEGER IERROR)</code>
MPI_Alltoallv	<code>int MPI_Alltoallv(void* sendbuf,int *sendcounts,int *sdispls,MPI_Datatype sendtype,void* recvbuf,int *recvcounts,int *rdispls,MPI_Datatype recvtype,MPI_Comm comm);</code>
MPI::Comm::Alltoallv	<code>void MPI::Comm::Alltoallv(const void* sendbuf, const int sendcounts[], const int sdispls[], const MPI::Datatype&amp; sendtype, void* recvbuf, const int recvcounts[], const int rdispls[], const MPI::Datatype&amp; recvtype) const;</code>
MPI_ALLTOALLV	<code>MPI_ALLTOALLV(CHOICE SENDBUF,INTEGER SENDCOUNTS(*),INTEGER SDISPLS(*),INTEGER SENDTYPE,CHOICE RECVBUF,INTEGER RECVCOUNTS(*),INTEGER RDISPLS(*),INTEGER RECVTYPE,INTEGER COMM,INTEGER IERROR)</code>
MPI_Alltoallw	<code>int MPI_Alltoallw(void* sendbuf, int sendcounts[], int sdispls[], MPI_Datatype sendtypes[], void* recvbuf, int recvcounts[], int rdispls[], MPI_Datatype recvtypes[], MPI_Comm comm);</code>
MPI::Comm::Alltoallw	<code>void MPI::Comm::Alltoallw(const void* sendbuf, const int sendcounts[], const int sdispls[], const MPI::Datatype sendtypes[], void* recvbuf, const int recvcounts[], const int rdispls[], const MPI::Datatype recvtypes[]) const;</code>

Table 26. Bindings for collective communication (continued)

Subroutine name: C C++ FORTRAN	Binding: C C++ FORTRAN
MPI_ALLTOALLW	MPI_ALLTOALLW(CHOICE SENDBUF(*), INTEGER SENDCOUNTS(*), INTEGER SDISPLS(*), INTEGER SENDTYPES(*), CHOICE RECVBUFF, INTEGER RECVCOUNTS(*), INTEGER RDISPLS(*), INTEGER RECVTYPES(*), INTEGER COMM, INTEGER IERROR)
MPI_Barrier	int MPI_Barrier(MPI_Comm comm);
MPI::Comm::Barrier()	void MPI::Comm::Barrier() const;
MPI_BARRIER	MPI_BARRIER(INTEGER COMM,INTEGER IERROR)
MPI_Bcast	int MPI_Bcast(void* buffer,int count,MPI_Datatype datatype,int root,MPI_Comm comm);
MPI::Comm::Bcast	void MPI::Comm::Bcast(void* buffer, int count, const MPI::Datatype& datatype, int root) const;
MPI_BCAST	MPI_BCAST(CHOICE BUFFER,INTEGER COUNT,INTEGER DATATYPE,INTEGER ROOT,INTEGER COMM,INTEGER IERROR)
MPI_Exscan	int MPI_Exscan(void *sendbuf, void *recvbuf, int count, MPI_Datatype datatype, MPI_Op op, MPI_Comm comm);
MPI::Intracomm::Exscan	void MPI::Intracomm::Exscan(const void* sendbuf, void* recvbuf, int count, const MPI::Datatype& datatype, const MPI::Op& op) const;
MPI_EXSCAN	MPI_EXSCAN(CHOICE SENDBUFF, CHOICE RECVBUFF, INTEGER COUNT, INTEGER DATATYPE, INTEGER OP, INTEGER COMM, INTEGER IERROR)
MPI_Gather	int MPI_Gather(void* sendbuf,int sendcount,MPI_Datatype sendtype,void* recvbuf,int recvcnt,MPI_Datatype recvttype,int root,MPI_Comm comm);
MPI::Comm::Gather	void MPI::Comm::Gather(const void* sendbuf, int sendcount, const MPI::Datatype& sendtype, void* recvbuf, int recvcnt, const MPI::Datatype& recvttype, int root) const;
MPI_GATHER	MPI_GATHER(CHOICE SENDBUFF,INTEGER SENDCOUNT,INTEGER SENDTYPE,CHOICE RECVBUFF,INTEGER RECVCOUNT,INTEGER RECVTYPE,INTEGER ROOT,INTEGER COMM,INTEGER IERROR)
MPI_Gatherv	int MPI_Gatherv(void* sendbuf,int sendcount,MPI_Datatype sendtype,void* recvbuf,int *recvcnts,int *displs,MPI_Datatype recvttype,int root,MPI_Comm comm);
MPI::Comm::Gatherv	void MPI::Comm::Gatherv(const void* sendbuf, int sendcount, const MPI::Datatype& sendtype, void* recvbuf, const int recvcnts[], const int displs[], const MPI::Datatype& recvttype, int root) const;
MPI_GATHERV	MPI_GATHERV(CHOICE SENDBUFF,INTEGER SENDCOUNT,INTEGER SENDTYPE,CHOICE RECVBUFF,INTEGER RECVCOUNTS(*),INTEGER DISPLS(*),INTEGER RECVTYPE,INTEGER ROOT,INTEGER COMM,INTEGER IERROR)
MPI_Op_create	int MPI_Op_create(MPI_User_function *function, int commute, MPI_Op *op);
MPI::Op::Init	void MPI::Op::Init(MPI::User_function *func, bool commute);
MPI_OP_CREATE	MPI_OP_CREATE(EXTERNAL FUNCTION,INTEGER COMMUTE,INTEGER OP,INTEGER IERROR)

Table 26. Bindings for collective communication (continued)

<b>Subroutine name:</b> C C++ FORTRAN	<b>Binding:</b> C C++ FORTRAN
MPI_Op_free	int MPI_Op_free(MPI_Op *op);
MPI::Op::Free	void MPI::Op::Free();
MPI_OP_FREE	MPI_OP_FREE(INTEGER OP,INTEGER IERROR)
MPI_Reduce	int MPI_Reduce(void* sendbuf,void* recbuf,int count,MPI_Datatype datatype,MPI_Op op,int root,MPI_Comm comm);
MPI::Comm::Reduce	void MPI::Comm::Reduce(const void* sendbuf, void* recbuf, int count, const MPI::Datatype& datatype, const MPI::Op& op, int root) const;
MPI_REDUCE	MPI_REDUCE(CHOICE SENDBUF,CHOICE RECVBUF,INTEGER COUNT,INTEGER DATATYPE,INTEGER OP,INTEGER ROOT,INTEGER COMM,INTEGER IERROR)
MPI_Reduce_scatter	int MPI_Reduce_scatter(void* sendbuf,void* recbuf,int *recvcnts,MPI_Datatype datatype,MPI_Op op,MPI_Comm comm);
MPI::Comm::Reduce_scatter	void MPI::Comm::Reduce_scatter(const void* sendbuf, void* recbuf, int recvcnts[], const MPI::Datatype& datatype, const MPI::Op& op) const;
MPI_REDUCE_SCATTER	MPI_REDUCE_SCATTER(CHOICE SENDBUF,CHOICE RECVBUF,INTEGER RECVCOUNTS(*),INTEGER DATATYPE,INTEGER OP,INTEGER COMM,INTEGER IERROR)
MPI_Scan	int MPI_Scan(void* sendbuf,void* recbuf,int count,MPI_Datatype datatype,MPI_Op op,MPI_Comm comm);
MPI::Intracomm::Scan	void MPI::Intracomm::Scan(const void *sendbuf, void *recbuf, int count, const MPI::Datatype& datatype, const MPI::Op& op) const;
MPI_SCAN	MPI_SCAN(CHOICE SENDBUF,CHOICE RECVBUF,INTEGER COUNT,INTEGER DATATYPE,INTEGER OP,INTEGER COMM,INTEGER IERROR)
MPI_Scatter	int MPI_Scatter(void* sendbuf,int sendcount,MPI_Datatype sendtype,void* recbuf,int recvcount,MPI_Datatype recvtype,int root MPI_Comm comm);
MPI::Comm::Scatter	void MPI::Comm::Scatter(const void* sendbuf, int sendcount, const MPI::Datatype& sendtype, void* recbuf, int recvcount, const MPI::Datatype& recvtype, int root) const;
MPI_SCATTER	MPI_SCATTER(CHOICE SENDBUF,INTEGER SENDCOUNT,INTEGER SENDTYPE,CHOICE RECVBUF,INTEGER RECVCOUNT,INTEGER RECVTYP,INTEGER ROOT,INTEGER COMM,INTEGER IERROR)
MPI_Scatterv	int MPI_Scatterv(void* sendbuf,int *sendcounts,int *displs,MPI_Datatype sendtype,void* recbuf,int recvcount,MPI_Datatype recvtype,int root,MPI_Comm comm);
MPI::Comm::Scatterv	void MPI::Comm::Scatterv(const void* sendbuf, const int sendcounts[], const int displs[], const MPI::Datatype& sendtype, void* recbuf, int recvcount, const MPI::Datatype& recvtype, int root) const;
MPI_SCATTERV	MPI_SCATTERV(CHOICE SENDBUF,INTEGER SENDCOUNTS(*),INTEGER DISPLS(*),INTEGER SENDTYPE,CHOICE RECVBUF,INTEGER RECVCOUNT,INTEGER RECVTYP,INTEGER ROOT,INTEGER COMM,INTEGER IERROR)

## Bindings for communicators

Table 27 lists the bindings for communicator subroutines.

Table 27. Bindings for communicators

Subroutine name: C C++ FORTRAN	Binding: C C++ FORTRAN
MPI_Attr_delete	int MPI_Attr_delete(MPI_Comm comm,int keyval);
(none)	(none)
MPI_ATTR_DELETE	MPI_ATTR_DELETE(INTEGER COMM,INTEGER KEYVAL,INTEGER IERROR)
MPI_Attr_get	int MPI_Attr_get(MPI_Comm comm,int keyval,void *attribute_val, int *flag);
(none)	(none)
MPI_ATTR_GET	MPI_ATTR_GET(INTEGER COMM,INTEGER KEYVAL,INTEGER ATTRIBUTE_VAL, LOGICAL FLAG,INTEGER IERROR)
MPI_Attr_put	int MPI_Attr_put(MPI_Comm comm,int keyval,void* attribute_val);
(none)	(none)
MPI_ATTR_PUT	MPI_ATTR_PUT(INTEGER COMM,INTEGER KEYVAL,INTEGER ATTRIBUTE_VAL, INTEGER IERROR)
(none)	(none)
MPI::Comm::Clone	MPI::Cartcomm& MPI::Cartcomm::Clone() const;  MPI::Graphcomm& MPI::Graphcomm::Clone() const;  MPI::Intercomm& MPI::Intercomm::Clone() const;  MPI::Intracomm& MPI::Intracomm::Clone() const;
(none)	(none)
MPI_Comm_compare	int MPI_Comm_compare(MPI_Comm comm1,MPI_Comm comm2,int *result);
MPI::Comm::Compare	int MPI::Comm::Compare(const MPI::Comm& comm1, const MPI::Comm& comm2);
MPI_COMM_COMPARE	MPI_COMM_COMPARE(INTEGER COMM1,INTEGER COMM2,INTEGER RESULT,INTEGER IERROR)
MPI_Comm_create	int MPI_Comm_create(MPI_Comm comm_in, MPI_Group group, MPI_Comm *comm_out);
MPI::Intercomm::Create MPI::Intracomm::Create	MPI::Intercomm MPI::Intercomm::Create(const MPI::Group& group) const;  MPI::Intracomm MPI::Intracomm::Create(const MPI::Group& group) const;
MPI_COMM_CREATE	MPI_COMM_CREATE(INTEGER COMM_IN, INTEGER GROUP, INTEGER COMM_OUT,INTEGER IERROR)
MPI_Comm_create_errhandler	int MPI_Comm_create_errhandler (MPI_Comm_errhandler_fn *function, MPI_Errhandler *errhandler);
MPI::Comm::Create_errhandler	static MPI::Errhandler MPI::Comm::Create_errhandler(MPI::Comm::Errhandler_fn* function);

Table 27. Bindings for communicators (continued)

Subroutine name: C C++ FORTRAN	Binding: C C++ FORTRAN
MPI_COMM_CREATE_ERRHANDLER	MPI_COMM_CREATE_ERRHANDLER(EXTERNAL FUNCTION, INTEGER ERRHANDLER, INTEGER IERROR)
MPI_Comm_create_keyval	int MPI_Comm_create_keyval (MPI_Comm_copy_attr_function *comm_copy_attr_fn, MPI_Comm_delete_attr_function *comm_delete_attr_fn, int *comm_keyval, void *extra_state);
MPI::Comm::Create_keyval	int MPI::Comm::Create_keyval(MPI::Comm::Copy_attr_function* comm_copy_attr_fn, MPI::Comm::Delete_attr_function* comm_delete_attr_fn, void* extra_state);
MPI_COMM_CREATE_KEYVAL	MPI_COMM_CREATE_KEYVAL(EXTERNAL COMM_COPY_ATTR_FN, EXTERNAL COMM_DELETE_ATTR_FN, INTEGER COMM_KEYVAL, INTEGER EXTRA_STATE, INTEGER IERROR)
MPI_Comm_delete_attr	int MPI_Comm_delete_attr (MPI_Comm comm, int comm_keyval);
MPI::Comm::Delete_attr	void MPI::Comm::Delete_attr(int comm_keyval);
MPI_COMM_DELETE_ATTR	MPI_COMM_DELETE_ATTR(INTEGER COMM, INTEGER COMM_KEYVAL, INTEGER IERROR)
MPI_Comm_dup	int MPI_Comm_dup(MPI_Comm comm, MPI_Comm *newcomm);
MPI::Cartcomm::Dup	MPI::Cartcomm MPI::Cartcomm::Dup() const;
MPI::Graphcomm::Dup	MPI::Graphcomm MPI::Graphcomm::Dup() const;
MPI::Intercomm::Dup	MPI::Intercomm MPI::Intercomm::Dup() const;
MPI::Intracomm::Dup	MPI::Intracomm MPI::Intracomm::Dup() const;
MPI_COMM_DUP	MPI_COMM_DUP(INTEGER COMM, INTEGER NEWCOMM, INTEGER IERROR)
MPI_Comm_free	int MPI_Comm_free(MPI_Comm *comm);
MPI::Comm::Free	void MPI::Comm::Free(void);
MPI_COMM_FREE	MPI_COMM_FREE(INTEGER COMM, INTEGER IERROR)
MPI_Comm_free_keyval	int MPI_Comm_free_keyval (int *comm_keyval);
MPI::Comm::Free_keyval	void MPI::Comm::Free_keyval(int& comm_keyval);
MPI_COMM_FREE_KEYVAL	MPI_COMM_FREE_KEYVAL(INTEGER COMM_KEYVAL, INTEGER IERROR)
MPI_Comm_get_attr	int MPI_Comm_get_attr (MPI_Comm comm, int comm_keyval, void *attribute_val, int *flag);
MPI::Comm::Get_attr	bool MPI::Comm::Get_attr(int comm_keyval, void* attribute_val) const;
MPI_COMM_GET_ATTR	MPI_COMM_GET_ATTR(INTEGER COMM, INTEGER COMM_KEYVAL, INTEGER ATTRIBUTE_VAL, LOGICAL FLAG, INTEGER IERROR)
MPI_Comm_get_errhandler	int MPI_Comm_get_errhandler (MPI_Comm comm, MPI_Errhandler *errhandler);
MPI::Comm::Get_errhandler	MPI::Errhandler MPI::Comm::Get_errhandler() const;
MPI_COMM_GET_ERRHANDLER	MPI_COMM_GET_ERRHANDLER(INTEGER COMM, INTEGER ERRHANDLER, INTEGER IERROR)
MPI_Comm_rank	int MPI_Comm_rank(MPI_Comm comm, int *rank);

Table 27. Bindings for communicators (continued)

Subroutine name: C C++ FORTRAN	Binding: C C++ FORTRAN
MPI::Comm::Get_rank	int MPI::Comm::Get_rank() <i>const</i> ;
MPI_COMM_RANK	MPI_COMM_RANK(INTEGER COMM,INTEGER RANK,INTEGER IERROR)
MPI_Comm_remote_group	int MPI_Comm_remote_group(MPI_Comm comm,MPI_group *group);
MPI::Intercomm::Get_remote_group	MPI::Group MPI::Intercomm::Get_remote_group() <i>const</i> ;
MPI_COMM_REMOTE_GROUP	MPI_COMM_REMOTE_GROUP(INTEGER COMM,MPI_GROUP GROUP,INTEGER IERROR)
MPI_Comm_remote_size	int MPI_Comm_remote_size(MPI_Comm comm,int *size);
MPI::Intercomm::Get_remote_size	int MPI::Intercomm::Get_remote_size() <i>const</i> ;
MPI_COMM_REMOTE_SIZE	MPI_COMM_REMOTE_SIZE(INTEGER COMM,INTEGER SIZE,INTEGER IERROR)
MPI_Comm_set_attr	int MPI_Comm_set_attr (MPI_Comm comm, int comm_keyval, void *attribute_val);
MPI::Comm::Set_attr	void MPI::Comm::Set_attr(int comm_keyval, const void* attribute_val) <i>const</i> ;
MPI_COMM_SET_ATTR	MPI_COMM_SET_ATTR(INTEGER COMM, INTEGER COMM_KEYVAL, INTEGER ATTRIBUTE_VAL, INTEGER IERROR)
MPI_Comm_set_errhandler	int MPI_Comm_set_errhandler (MPI_Comm comm, MPI_Errhandler *errhandler);
MPI::Comm::Set_errhandler	void MPI::Comm::Set_errhandler(const MPI::Errhandler& errhandler);
MPI_COMM_SET_ERRHANDLER	MPI_COMM_SET_ERRHANDLER(INTEGER COMM, INTEGER ERRHANDLER, INTEGER IERROR)
MPI_Comm_size	int MPI_Comm_size(MPI_Comm comm,int *size);
MPI::Comm::Get_size	int MPI::Comm::Get_size() <i>const</i> ;
MPI_COMM_SIZE	MPI_COMM_SIZE(INTEGER COMM,INTEGER SIZE,INTEGER IERROR)
MPI_Comm_split	int MPI_Comm_split(MPI_Comm comm_in, int color, int key, MPI_Comm *comm_out);
MPI::Intercomm::Split	MPI::Intercomm MPI::Intercomm::Split(int color, int key) <i>const</i> ;
MPI::Intracomm::Split	MPI::Intracomm MPI::Intracomm::Split(int color, int key) <i>const</i> ;
MPI_COMM_SPLIT	MPI_COMM_SPLIT(INTEGER COMM_IN, INTEGER COLOR, INTEGER KEY, INTEGER COMM_OUT, INTEGER IERROR)
MPI_Comm_test_inter	int MPI_Comm_test_inter(MPI_Comm comm,int *flag);
MPI::Comm::Is_inter	bool MPI::Comm::Is_inter() <i>const</i> ;
MPI_COMM_TEST_INTER	MPI_COMM_TEST_INTER(INTEGER COMM,LOGICAL FLAG,INTEGER IERROR)
MPI_Intercomm_create	int MPI_Intercomm_create(MPI_Comm local_comm,int local_leader, MPI_Comm peer_comm,int remote_leader,int tag,MPI_Comm *newintercom);
MPI::Intracomm::Create_intercomm	MPI::Intercomm MPI::Intracomm::Create_intercomm(int local_leader, const MPI::Comm& peer_comm, int remote_leader, int tag) <i>const</i> ;

Table 27. Bindings for communicators (continued)

Subroutine name:	Binding:
C	C
C++	C++
FORTRAN	FORTRAN
MPI_INTERCOMM_CREATE	MPI_INTERCOMM_CREATE(INTEGER LOCAL_COMM,INTEGER LOCAL_LEADER, INTEGER PEER_COMM,INTEGER REMOTE_LEADER,INTEGER TAG, INTEGER NEWINTERCOM,INTEGER IERROR)
MPI_Intercomm_merge	int MPI_Intercomm_merge(MPI_Comm intercomm,int high, MPI_Comm *newintracomm);
MPI::Intercomm::Merge	MPI::Intracomm MPI::Intercomm::Merge(bool high);
MPI_INTERCOMM_MERGE	MPI_INTERCOMM_MERGE(INTEGER INTERCOMM,INTEGER HIGH, INTEGER NEWINTRACOMM,INTEGER IERROR)
MPI_Keyval_create	int MPI_Keyval_create(MPI_Copy_function *copy_fn, MPI_Delete_function *delete_fn,int *keyval, void* extra_state);
(none)	(none)
MPI_KEYVAL_CREATE	MPI_KEYVAL_CREATE(EXTERNAL COPY_FN,EXTERNAL DELETE_FN, INTEGER KEYVAL,INTEGER EXTRA_STATE,INTEGER IERROR)
MPI_Keyval_free	int MPI_Keyval_free(int *keyval);
(none)	(none)
MPI_KEYVAL_FREE	MPI_KEYVAL_FREE(INTEGER KEYVAL,INTEGER IERROR)

## Bindings for conversion functions

Table 28 lists the C bindings for conversion functions. These functions do not have C++ or FORTRAN bindings.

Table 28. Bindings for conversion functions

Function name:	C binding:
MPI_Comm_c2f	MPI_Fint MPI_Comm_c2f(MPI_Comm comm);
MPI_Comm_f2c	MPI_Comm MPI_Comm_f2c(MPI_Fint comm);
MPI_Errhandler_c2f	MPI_Fint MPI_Errhandler_c2f(MPI_Errhandler errhandler);
MPI_Errhandler_f2c	MPI_Errhandler MPI_Errhandler_f2c(MPI_Fint errorhandler);
MPI_File_c2f	MPI_Fint MPI_File_c2f(MPI_File file);
MPI_File_f2c	MPI_File MPI_File_f2c(MPI_Fint file);
MPI_Group_c2f	MPI_Fint MPI_Group_c2f(MPI_Group group);
MPI_Group_f2c	MPI_Group MPI_Group_f2c(MPI_Fint group);
MPI_Info_c2f	MPI_Fint MPI_Info_c2f(MPI_Info info);
MPI_Info_f2c	MPI_Info MPI_Info_f2c(MPI_Fint file);
MPI_Op_c2f	MPI_Fint MPI_Op_c2f(MPI_Op op);
MPI_Op_f2c	MPI_Op MPI_Op_f2c(MPI_Fint op);
MPI_Request_c2f	MPI_Fint MPI_Request_c2f(MPI_Request request);
MPI_Request_f2c	MPI_Request MPI_Request_f2c(MPI_Fint request);
MPI_Status_c2f	int MPI_Status_c2f(MPI_Status *c_status, MPI_Fint *f_status);
MPI_Status_f2c	int MPI_Status_f2c(MPI_Fint *f_status, MPI_Status *c_status);

Table 28. Bindings for conversion functions (continued)

Function name:	C binding:
MPI_Type_c2f	MPI_Fint MPI_Type_c2f(MPI_Type datatype);
MPI_Type_f2c	MPI_Type MPI_Type_f2c(MPI_Fint datatype);
MPI_Win_c2f	MPI_Fint MPI_Win_c2f(MPI_Win win);
MPI_Win_f2c	MPI_Win MPI_Win_f2c(MPI_Fint win);

## Bindings for derived datatypes

Table 29 lists the bindings for derived datatype subroutines.

Table 29. Bindings for derived datatypes

Subroutine name:	Binding:
C	C
C++	C++
FORTRAN	FORTRAN
MPI_Address	int MPI_Address(void* location, MPI_Aint *address);
(none)	(none)
MPI_ADDRESS	MPI_ADDRESS(CHOICE LOCATION, INTEGER ADDRESS, INTEGER IERROR)
MPI_Get_address	int MPI_Get_address(void *location, MPI_Aint *address);
MPI::Get_address	MPI::Aint MPI::Get_address(void* location);
MPI_GET_ADDRESS	MPI_GET_ADDRESS(CHOICE LOCATION(*), INTEGER(KIND=MPI_ADDRESS_KIND) ADDRESS, INTEGER IERROR)
MPI_Get_elements	int MPI_Get_elements(MPI_Status *status, MPI_Datatype datatype, int *count);
MPI::Status::Get_elements	int MPI::Status::Get_elements(const MPI::Datatype& datatype) const;
MPI_GET_ELEMENTS	MPI_GET_ELEMENTS(INTEGER STATUS(MPI_STATUS_SIZE), INTEGER DATATYPE, INTEGER COUNT, INTEGER IERROR)
MPI_Pack	int MPI_Pack(void* inbuf, int incount, MPI_Datatype datatype, void *outbuf, int outsize, int *position, MPI_Comm comm);
MPI::Datatype::Pack	void MPI::Datatype::Pack(const void* inbuf, int incount, void* outbuf, int outsize, int& position, const MPI::Comm& comm) const;
MPI_PACK	MPI_PACK(CHOICE INBUF, INTEGER INCOUNT, INTEGER DATATYPE, CHOICE OUTBUF, INTEGER OUTSIZE, INTEGER POSITION, INTEGER COMM, INTEGER IERROR)
MPI_Pack_external	int MPI_Pack_external(char *datarep, void *inbuf, int incount, MPI_Datatype datatype, void *outbuf, MPI_Aint outsize, MPI_Aint *position);
MPI::Datatype::Pack_external	void MPI::Datatype::Pack_external(const char* datarep, const void* inbuf, int incount, void* outbuf, MPI::Aint outsize, MPI_Aint& position) const;
MPI_PACK_EXTERNAL	MPI_PACK_EXTERNAL(CHARACTER(*) DATAREP, CHOICE INBUF(*), INTEGER INCOUNT, INTEGER DATATYPE, CHOICE OUTBUF(*), INTEGER(KIND=MPI_ADDRESS_KIND) OUTSIZE, INTEGER(KIND=MPI_ADDRESS_KIND) POSITION, INTEGER IERROR)

Table 29. Bindings for derived datatypes (continued)

Subroutine name: C C++ FORTRAN	Binding: C C++ FORTRAN
MPI_Pack_external_size	int MPI_Pack_external_size(char *datarep, int incount, MPI_Datatype datatype, MPI_Aint *size);
MPI::Datatype::Pack_external_size	MPI::Aint MPI::Datatype::Pack_external_size(const char* datarep, int incount) const;
MPI_PACK_EXTERNAL_SIZE	MPI_PACK_EXTERNAL_SIZE(CHARACTER*(*) DATAREP, INTEGER INCOUNT, INTEGER DATATYPE, INTEGER(KIND=MPI_ADDRESS_KIND) SIZE, INTEGER IERROR)
MPI_Pack_size	int MPI_Pack_size(int incount, MPI_Datatype datatype, MPI_Comm comm, int *size);
MPI::Datatype::Pack_size	int MPI::Datatype::Pack_size(int incount, const MPI::Comm& comm) const;
MPI_PACK_SIZE	MPI_PACK_SIZE(INTEGER INCOUNT, INTEGER DATATYPE, INTEGER COMM, INTEGER SIZE, INTEGER IERROR)
(none)	(none)
(none)	(none)
MPI_SIZEOF	MPI_SIZEOF(CHOICE X, INTEGER SIZE, INTEGER IERROR)
MPI_Type_commit	int MPI_Type_commit(MPI_Datatype *datatype);
MPI::Datatype::Commit	void MPI::Datatype::Commit();
MPI_TYPE_COMMIT	MPI_TYPE_COMMIT(INTEGER DATATYPE, INTEGER IERROR)
MPI_Type_contiguous	int MPI_Type_contiguous(int count, MPI_Datatype oldtype, MPI_Datatype *newtype);
MPI::Datatype::Create_contiguous	MPI::Datatype MPI::Datatype::Create_contiguous(int count) const;
MPI_TYPE_CONTIGUOUS	MPI_TYPE_CONTIGUOUS(INTEGER COUNT, INTEGER OLDTYPE, INTEGER NEWTYPE, INTEGER IERROR)
MPI_Type_create_darray	int MPI_Type_create_darray (int size, int rank, int ndims, int array_of_gsizes[], int array_of_distrib[], int array_of_dargs[], int array_of_psize[], int order, MPI_Datatype oldtype, MPI_Datatype *newtype);
MPI::Datatype::Create_darray	MPI::Datatype MPI::Datatype::Create_darray(int size, int rank, int ndims, const int array_of_gsizes[], const int array_of_distrib[], const int array_of_dargs[], const int array_of_psize[], int order) const;
MPI_TYPE_CREATE_DARRAY	MPI_TYPE_CREATE_DARRAY (INTEGER SIZE, INTEGER RANK, INTEGER NDIMS, INTEGER ARRAY_OF_GSIZES(*), INTEGER ARRAY_OF_DISTRIBS(*), INTEGER ARRAY_OF_DARGS(*), INTEGER ARRAY_OF_PSIZE(*), INTEGER ORDER, INTEGER OLDTYPE, INTEGER NEWTYPE, INTEGER IERROR)
MPI_Type_create_f90_complex	int MPI_Type_create_f90_complex(int p, int r, MPI_Datatype *newtype);
MPI::Datatype::Create_f90_complex	static MPI::Datatype MPI::Datatype::Create_f90_complex(int p, int r);
MPI_TYPE_CREATE_F90_COMPLEX	MPI_TYPE_CREATE_F90_COMPLEX(INTEGER P, INTEGER R, INTEGER NEWTYPE, INTEGER IERROR)
MPI_Type_create_f90_integer	int MPI_Type_create_f90_integer(int r, MPI_Datatype *newtype);
MPI::Datatype::Create_f90_integer	static MPI::Datatype MPI::Datatype::Create_f90_integer(int r);

Table 29. Bindings for derived datatypes (continued)

Subroutine name: C C++ FORTRAN	Binding: C C++ FORTRAN
MPI_TYPE_CREATE_F90_INTEGER	MPI_TYPE_CREATE_F90_INTEGER(INTEGER R, INTEGER NEWTYPE, INTEGER IERROR)
MPI_Type_create_f90_real	int MPI_Type_create_f90_real(int p, int r, MPI_Datatype *newtype);
MPI::Datatype::Create_f90_real	static MPI::Datatype MPI::Datatype::Create_f90_real(int p, int r);
MPI_TYPE_CREATE_F90_REAL	MPI_TYPE_CREATE_F90_REAL(INTEGER P, INTEGER R, INTEGER NEWTYPE, INTEGER IERROR)
MPI_Type_create_hindexed	int MPI_Type_create_hindexed(int count, int array_of_blocklengths[], MPI_Aint array_of_displacements[], MPI_Datatype oldtype, MPI_Datatype *newtype);
MPI::Datatype::Create_hindexed	MPI::Datatype MPI::Datatype::Create_hindexed(int count, const int array_of_blocklengths[], const MPI::Aint array_of_displacements[]) const;
MPI_TYPE_CREATE_HINDEXED	MPI_TYPE_CREATE_HINDEXED(INTEGER COUNT, INTEGER ARRAY_OF_BLOCKLENGTHS(*), INTEGER(KIND=MPI_ADDRESS_KIND) ARRAY_OF_DISPLACEMENTS(*), INTEGER OLDTYPE, INTEGER NEWTYPE, INTEGER IERROR)
MPI_Type_create_hvector	int MPI_Type_create_hvector(int count, int blocklength, MPI_Aint stride, MPI_Datatype oldtype, MPI_Datatype *newtype);
MPI::Datatype::Create_hvector	MPI::Datatype MPI::Datatype::Create_hvector(int count, int blocklength, MPI::Aint stride) const;
MPI_TYPE_CREATE_HVECTOR	MPI_TYPE_CREATE_HVECTOR(INTEGER COUNT, INTEGER BLOCKLENGTH, INTEGER(KIND=MPI_ADDRESS_KIND) STRIDE, INTEGER OLDTYPE, INTEGER NEWTYPE, INTEGER IERROR)
MPI_Type_create_indexed_block	int MPI_Type_create_indexed_block(int count, int blocklength, int array_of_displacements[], MPI_Datatype oldtype, MPI_datatype *newtype);
MPI::Datatype::Create_indexed_block	MPI::Datatype MPI::Datatype::Create_indexed_block(int count, int blocklength, const int array_of_displacements[]) const;
MPI_TYPE_CREATE_INDEXED_BLOCK	MPI_TYPE_CREATE_INDEXED_BLOCK(INTEGER COUNT, INTEGER BLOCKLENGTH, INTEGER ARRAY_OF_DISPLACEMENTS(*), INTEGER OLDTYPE, INTEGER NEWTYPE, INTEGER IERROR)
MPI_Type_create_keyval	int MPI_Type_create_keyval (MPI_Type_copy_attr_function *type_copy_attr_fn, MPI_Type_delete_attr_function *type_delete_attr_fn, int *type_keyval, void *extra_state);
MPI::Datatype::Create_keyval	int MPI::Datatype::Create_keyval(MPI::Datatype::Copy_attr_function* type_copy_attr_fn, MPI::Datatype::Delete_attr_function* type_delete_attr_fn, void* extra_state);
MPI_TYPE_CREATE_KEYVAL	MPI_TYPE_CREATE_KEYVAL(EXTERNAL TYPE_COPY_ATTR_FN, EXTERNAL TYPE_DELETE_ATTR_FN, INTEGER TYPE_KEYVAL, INTERGER EXTRA_STATE, INTEGER IERROR)
MPI_Type_create_resized	int MPI_Type_create_resized(MPI_Datatype oldtype, MPI_Aint lb, MPI_Aint extent, MPI_Datatype *newtype);
MPI::Datatype::Create_resized	MPI::Datatype MPI::Datatype::Create_resized(const MPI::Aint lb, const MPI::Aint extent) const;

Table 29. Bindings for derived datatypes (continued)

Subroutine name: C C++ FORTRAN	Binding: C C++ FORTRAN
MPI_TYPE_CREATE_RESIZED	MPI_TYPE_CREATE_RESIZED(INTEGER OLDTYPE, INTEGER LB, INTEGER(KIND=MPI_ADDRESS_KIND) EXTENT, INTEGER NEWTYPE, INTEGER IERROR)
MPI_Type_create_struct	int MPI_Type_create_struct(int count, int array_of_blocklengths[], MPI_Aint array_of_displacements[], MPI_Datatype array_of_types[], MPI_datatype *newtype);
MPI::Datatype::Create_struct	static MPI::Datatype MPI::Datatype::Create_struct(int count, const int array_of_blocklengths[], const MPI::Aint array_of_displacements[], const MPI::Datatype array_of_types[]);
MPI_TYPE_CREATE_STRUCT	MPI_TYPE_CREATE_STRUCT(INTEGER COUNT, INTEGER ARRAY_OF_BLOCKLENGTHS(*), INTEGER(KIND=MPI_ADDRESS_KIND) ARRAY_OF_DISPLACEMENTS(*), INTEGER ARRAY_OF_TYPES(*), INTEGER NEWTYPE, INTEGER IERROR)
MPI_Type_create_subarray	int MPI_Type_create_subarray (int ndims,int array_of_sizes[], int array_of_subsizes[],int array_of_starts[], int order,MPI_Datatype oldtype,MPI_Datatype *newtype);
MPI::Datatype::Create_subarray	MPI::Datatype MPI::Datatype::Create_subarray(int ndims, const int array_of_sizes[], const int array_of_subsizes[], const int array_of_starts[], int order) const;
MPI_TYPE_CREATE_SUBARRAY	MPI_TYPE_CREATE_SUBARRAY (INTEGER NDIMS,INTEGER ARRAY_OF_SUBSIZES(*), INTEGER ARRAY_OF_SIZES(*),INTEGER ARRAY_OF_STARTS(*), INTEGER ORDER,INTEGER OLDTYPE,INTEGER NEWTYPE,INTEGER IERROR)
MPI_Type_delete_attr	int MPI_Type_delete_attr (MPI_Datatype type, int type_keyval);
MPI::Datatype::Delete_attr	void MPI::Datatype::Delete_attr(int type_keyval);
MPI_TYPE_DELETE_ATTR	MPI_TYPE_DELETE_ATTR(INTEGER TYPE, INTEGER TYPE_KEYVAL, INTEGER IERROR)
MPI_Type_dup	int MPI_Type_dup (MPI_Datatype type, MPI_Datatype *newtype);
MPI::Datatype::Dup	MPI::Datatype MPI::Datatype::Dup() const;
MPI_TYPE_DUP	MPI_TYPE_DUP(INTEGER TYPE, INTEGER NEWTYPE, INTEGER IERROR)
MPI_Type_extent	int MPI_Type_extent(MPI_Datatype datatype, int *extent);
(none)	(none)
MPI_TYPE_EXTENT	MPI_TYPE_EXTENT(INTEGER DATATYPE, INTEGER EXTENT, INTEGER IERROR)
MPI_Type_free	int MPI_Type_free(MPI_Datatype *datatype);
MPI::Datatype::Free	void MPI::Datatype::Free();
MPI_TYPE_FREE	MPI_TYPE_FREE(INTEGER DATATYPE,INTEGER IERROR)
MPI_Type_free_keyval	int MPI_Type_free_keyval (int *type_keyval);
MPI::Datatype::Free_keyval	void MPI::Datatype::Free_keyval(int& type_keyval);
MPI_TYPE_FREE_KEYVAL	MPI_TYPE_FREE_KEYVAL(INTEGER TYPE_KEYVAL, INTEGER IERROR)
MPI_Type_get_attr	int MPI_Type_get_attr (MPI_Datatype type, int type_keyval, void *attribute_val, int *flag);

Table 29. Bindings for derived datatypes (continued)

Subroutine name: C C++ FORTRAN	Binding: C C++ FORTRAN
MPI::Datatype::Get_attr	bool MPI::Datatype::Get_attr(int type_keyval, void* attribute_val) const;
MPI_TYPE_GET_ATTR	MPI_TYPE_GET_ATTR(INTEGER TYPE, INTEGER TYPE_KEYVAL, INTEGER ATTRIBUTE_VAL, LOGICAL FLAG, INTEGER IERROR)
MPI_Type_get_contents	int MPI_Type_get_contents(MPI_Datatype datatype, int *max_integers, int *max_addresses, int *max_datatypes, int array_of_integers[], int array_of_addresses[], int array_of_datatypes[]);
MPI::Datatype::Get_contents	void MPI::Datatype::Get_contents(int max_integers, int max_addresses, int max_datatypes, int array_of_integers[], MPI::Aint array_of_addresses[], MPI::Datatype array_of_datatypes[]) const;
MPI_TYPE_GET_CONTENTS	MPI_TYPE_GET_CONTENTS(INTEGER DATATYPE, INTEGER MAX_INTEGERS, INTEGER MAX_ADDRESSES, INTEGER MAX_DATATYPES, INTEGER ARRAY_of_INTEGERS(*), INTEGER ARRAY_OF_ADDRESSES(*), INTEGER ARRAY_of_DATATYPES(*), INTEGER IERROR)
MPI_Type_get_envelope	int MPI_Type_get_envelope(MPI_Datatype datatype, int *num_integers, int *num_addresses, int *num_datatypes, int *combiner);
MPI::Datatype::Get_envelope	void MPI::Datatype::Get_envelope(int& num_integers, int& num_addresses, int& num_datatypes, int& combiner) const;
MPI_TYPE_GET_ENVELOPE	MPI_TYPE_GET_ENVELOPE(INTEGER DATATYPE, INTEGER NUM_INTEGERS, INTEGER NUM_ADDRESSES, INTEGER NUM_DATATYPES, INTEGER COMBINER, INTEGER IERROR)
MPI_Type_get_extent	int MPI_Type_get_extent(MPI_Datatype datatype, MPI_Aint *lb, MPI_Aint *extent);
MPI::Datatype::Get_extent	void MPI::Datatype::Get_extent(MPI::Aint& lb, MPI::Aint& extent) const;
MPI_TYPE_GET_EXTENT	MPI_TYPE_GET_EXTENT(INTEGER DATATYPE, INTEGER(KIND=MPI_ADDRESS_KIND) LB, INTEGER(KIND=MPI_ADDRESS_KIND) EXTENT, INTEGER IERROR)
MPI_Type_get_true_extent	int MPI_Type_get_true_extent(MPI_Datatype datatype, MPI_Aint *true_lb, MPI_Aint *true_extent);
MPI::Datatype::Get_true_extent	void MPI::Datatype::Get_true_extent(MPI::Aint& true_lb, MPI::Aint& true_extent) const;
MPI_TYPE_GET_TRUE_EXTENT	MPI_TYPE_GET_TRUE_EXTENT(INTEGER DATATYPE, INTEGER TRUE_LB, INTEGER(KIND=MPI_ADDRESS_KIND) TRUE_EXTENT, INTEGER IERROR)
MPI_Type_hindexed	int MPI_Type_hindexed(int count, int *array_of_blocklengths, MPI_Aint *array_of_displacements, MPI_Datatype oldtype, MPI_Datatype *newtype);
(none)	(none)
MPI_TYPE_HINDEXED	MPI_TYPE_HINDEXED(INTEGER COUNT, INTEGER ARRAY_OF_BLOCKLENGTHS(*), INTEGER ARRAY_OF_DISPLACEMENTS(*), INTEGER OLDTYPE, INTEGER NEWTYPE, INTEGER IERROR)
MPI_Type_hvector	int MPI_Type_hvector(int count, int blocklength, MPI_Aint stride, MPI_Datatype oldtype, MPI_Datatype *newtype);
(none)	(none)

Table 29. Bindings for derived datatypes (continued)

Subroutine name: C C++ FORTRAN	Binding: C C++ FORTRAN
MPI_TYPE_HVECTOR	MPI_TYPE_HVECTOR(INTEGER COUNT, INTEGER BLOCKLENGTH, INTEGER STRIDE, INTEGER OLDTYPE, INTEGER NEWTYPE, INTEGER IERROR)
MPI_Type_indexed	int MPI_Type_indexed(int count, int *array_of_blocklengths, int *array_of_displacements, MPI_Datatype oldtype, MPI_Datatype *newtype);
MPI::Datatype::Create_indexed	MPI::Datatype MPI::Datatype::Create_indexed(int count, const int array_of_blocklengths[], const int array_of_displacements[]) const;
MPI_TYPE_INDEXED	MPI_TYPE_INDEXED(INTEGER COUNT, INTEGER ARRAY_OF_BLOCKLENGTHS(*), INTEGER ARRAY_OF_DISPLACEMENTS(*), INTEGER OLDTYPE, INTEGER NEWTYPE, INTEGER IERROR)
MPI_Type_lb	int MPI_Type_lb(MPI_Datatype datatype, int* displacement);
(none)	(none)
MPI_TYPE_LB	MPI_TYPE_LB(INTEGER DATATYPE, INTEGER DISPLACEMENT, INTEGER IERROR)
MPI_Type_match_size	int MPI_Type_match_size(int typeclass, int size, MPI_Datatype *type);
MPI::Datatype::Match_size	static MPI::Datatype MPI::Datatype::Match_size(int typeclass, int size);
MPI_TYPE_MATCH_SIZE	MPI_TYPE_MATCH_SIZE(INTEGER TYPECLASS, INTEGER SIZE, INTEGER TYPE, INTEGER IERROR)
MPI_Type_set_attr	int MPI_Type_set_attr (MPI_Datatype type, int type_keyval, void *attribute_val);
MPI::Datatype::Set_attr	void MPI::Datatype::Set_attr(int type_keyval, const void* attribute_val);
MPI_TYPE_SET_ATTR	MPI_TYPE_SET_ATTR(INTEGER TYPE, INTEGER TYPE_KEYVAL, INTEGER ATTRIBUTE_VAL, INTEGER IERROR)
MPI_Type_size	int MPI_Type_size(MPI_Datatype datatype, int *size);
MPI::Datatype::Get_size	int MPI::Datatype::Get_size() const;
MPI_TYPE_SIZE	MPI_TYPE_SIZE(INTEGER DATATYPE, INTEGER SIZE, INTEGER IERROR)
MPI_Type_struct	int MPI_Type_struct(int count, int *array_of_blocklengths, MPI_Aint *array_of_displacements, MPI_Datatype *array_of_types, MPI_Datatype *newtype);
(none)	(none)
MPI_TYPE_STRUCT	MPI_TYPE_STRUCT(INTEGER COUNT, INTEGER ARRAY_OF_BLOCKLENGTHS(*), INTEGER ARRAY_OF_DISPLACEMENTS(*), INTEGER ARRAY_OF_TYPES(*), INTEGER NEWTYPE, INTEGER IERROR)
MPI_Type_ub	int MPI_Type_ub(MPI_Datatype datatype, int* displacement);
(none)	(none)
MPI_TYPE_UB	MPI_TYPE_UB(INTEGER DATATYPE, INTEGER DISPLACEMENT, INTEGER IERROR)
MPI_Type_vector	int MPI_Type_vector(int count, int blocklength, int stride, MPI_Datatype oldtype, MPI_Datatype *newtype);
MPI::Datatype::Create_vector	MPI::Datatype MPI::Datatype::Create_vector(int count, int blocklength, int stride) const;

Table 29. Bindings for derived datatypes (continued)

Subroutine name: C C++ FORTRAN	Binding: C C++ FORTRAN
MPI_TYPE_VECTOR	MPI_TYPE_VECTOR(INTEGER COUNT, INTEGER BLOCKLENGTH, INTEGER STRIDE, INTEGER OLDTYPE, INTEGER NEWTYPE, INTEGER IERROR)
MPI_Unpack	int MPI_Unpack(void* inbuf,int insize,int *position,void *outbuf,int outcount,MPI_Datatype datatype,MPI_Comm comm);
MPI::Datatype::Unpack	void MPI::Datatype::Unpack(const void* inbuf, int insize, void* outbuf, int outcount, int& position, const MPI::Comm& comm) const;
MPI_UNPACK	MPI_UNPACK(CHOICE INBUF,INTEGER INSIZE,INTEGER POSITION,CHOICE OUTBUF,INTEGER OUTCOUNT,INTEGER DATATYPE,INTEGER COMM, INTEGER IERROR)
MPI_Unpack_external	int MPI_Unpack_external(char *datarep, void *inbuf, MPI_Aint insize, MPI_Aint *position, void *outbuf, int outcount, MPI_Datatype datatype);
MPI::Datatype::Unpack_external	void MPI::Datatype::Unpack_external(const char* datarep, const void* inbuf, MPI::Aint insize, MPI::Aint& position, void* outbuf, int outcount) const;
MPI_UNPACK_EXTERNAL	MPI_UNPACK_EXTERNAL(CHARACTER*(*) DATAREP, CHOICE INBUF(*), INTEGER(KIND=MPI_ADDRESS_KIND) INSIZE, INTEGER(KIND=MPI_ADDRESS_KIND) POSITION, CHOICE OUTBUF(*), INTEGER OUTCOUNT, INTEGER DATATYPE, INTEGER IERROR)

## Bindings for environment management

Table 30 lists the bindings for environment management subroutines.

Table 30. Bindings for environment management

Subroutine name: C C++ FORTRAN	Binding: C C++ FORTRAN
MPI_Abort	int MPI_Abort(MPI_Comm comm, int errorcode);
MPI::Comm::Abort	void MPI::Comm::Abort(int errorcode);
MPI_ABORT	MPI_ABORT(INTEGER COMM,INTEGER ERRORCODE,INTEGER IERROR)
MPI_Errhandler_create	int MPI_Errhandler_create(MPI_Handler_function *function, MPI_Errhandler *errhandler);
(none)	(none)
MPI_ERRHANDLER_CREATE	MPI_ERRHANDLER_CREATE(EXTERNAL FUNCTION,INTEGER ERRHANDLER, INTEGER IERROR)
MPI_Errhandler_free	int MPI_Errhandler_free(MPI_Errhandler *errhandler);
MPI::Errhandler::Free	void MPI::Errhandler::Free();
MPI_ERRHANDLER_FREE	MPI_ERRHANDLER_FREE(INTEGER ERRHANDLER,INTEGER IERROR)
MPI_Errhandler_get	int MPI_Errhandler_get(MPI_Comm comm,MPI_Errhandler *errhandler);

Table 30. Bindings for environment management (continued)

Subroutine name: C C++ FORTRAN	Binding: C C++ FORTRAN
(none)	(none)
MPI_ERRHANDLER_GET	MPI_ERRHANDLER_GET(INTEGER COMM,INTEGER ERRHANDLER,INTEGER IERROR)
MPI_Errhandler_set	int MPI_Errhandler_set(MPI_Comm comm,MPI_Errhandler errhandler);
(none)	(none)
MPI_ERRHANDLER_SET	MPI_ERRHANDLER_SET(INTEGER COMM,INTEGER ERRHANDLER,INTEGER IERROR)
MPI_Error_class	int MPI_Error_class(int errorcode, int *errorclass);
MPI::Get_error_class	int MPI::Get_error_class(int errorcode);
MPI_ERROR_CLASS	MPI_ERROR_CLASS(INTEGER ERRORCODE,INTEGER ERRORCLASS,INTEGER IERROR)
MPI_Error_string	int MPI_Error_string(int errorcode, char *string, int *resultlen);
MPI::Get_error_string	void MPI::Get_error_string(int errorcode, char* string, int& resultlen);
MPI_ERROR_STRING	MPI_ERROR_STRING(INTEGER ERRORCODE,CHARACTER STRING(*),INTEGER RESULTLEN,INTEGER IERROR)
MPI_File_create_errhandler	int MPI_File_create_errhandler (MPI_File_errhandler_fn *function, MPI_Errhandler *errhandler);
MPI::File::Create_errhandler	static MPI::Errhandler MPI::File::Create_errhandler(MPI::File::Errhandler_fn* function);
MPI_FILE_CREATE_ERRHANDLER	MPI_FILE_CREATE_ERRHANDLER(EXTERNAL FUNCTION,INTEGER ERRHANDLER, INTEGER IERROR)
MPI_File_get_errhandler	int MPI_File_get_errhandler (MPI_File file,MPI_Errhandler *errhandler);
MPI::File::Get_errhandler	MPI::Errhandler MPI::File::Get_errhandler() const;
MPI_FILE_GET_ERRHANDLER	MPI_FILE_GET_ERRHANDLER (INTEGER FILE,INTEGER ERRHANDLER, INTEGER IERROR)
MPI_File_set_errhandler	int MPI_File_set_errhandler (MPI_File fh, MPI_Errhandler errhandler);
MPI::File::Set_errhandler	void MPI::File::Set_errhandler(const MPI::Errhandler& errhandler);
MPI_FILE_SET_ERRHANDLER	MPI_FILE_SET_ERRHANDLER(INTEGER FH,INTEGER ERRHANDLER, INTEGER IERROR)
MPI_Finalize	int MPI_Finalize(void);
MPI::Finalize	void MPI::Finalize();
MPI_FINALIZE	MPI_FINALIZE(INTEGER IERROR)
MPI_Finalized	int MPI_Finalized(int *flag);
MPI::Is_finalized	bool MPI::Is_finalized();
MPI_FINALIZED	MPI_FINALIZED(LOGICAL FLAG, INTEGER IERROR)
MPI_Get_processor_name	int MPI_Get_processor_name(char *name,int *resultlen);
MPI::Get_processor_name	void MPI::Get_processor_name(char*& name, int& resultlen);
MPI_GET_PROCESSOR_NAME	MPI_GET_PROCESSOR_NAME(CHARACTER NAME(*),INTEGER RESULTLEN,INTEGER IERROR)
MPI_Get_version	int MPI_Get_version(int *version,int *subversion);
MPI::Get_version	void MPI::Get_version(int& version, int& subversion);

Table 30. Bindings for environment management (continued)

Subroutine name: C C++ FORTRAN	Binding: C C++ FORTRAN
MPI_GET_VERSION	MPI_GET_VERSION(INTEGER VERSION,INTEGER SUBVERSION,INTEGER IERROR)
MPI_Init	int MPI_Init(int *argc, char ***argv);
MPI::Init	void MPI::Init(int& argc, char**& argv); void MPI::Init();
MPI_INIT	MPI_INIT(INTEGER IERROR)
MPI_Init_thread	int MPI_Init_thread(int *argc, char *((*argv)[1]), int required, int *provided);
MPI::Init_thread	int MPI::Init_thread(int& argc, char**& argv, int required); int MPI::Init_thread(int required);
MPI_INIT_THREAD	MPI_INIT_THREAD(INTEGER REQUIRED, INTEGER PROVIDED, INTEGER IERROR)
MPI_Initialized	int MPI_Initialized(int *flag);
MPI::Is_initialized	bool MPI::Is_initialized();
MPI_INITIALIZED	MPI_INITIALIZED(INTEGER FLAG,INTEGER IERROR)
MPI_Is_thread_main	int MPI_Is_thread_main(int *flag);
MPI::Is_thread_main	bool MPI::Is_thread_main();
MPI_IS_THREAD_MAIN	MPI_IS_THREAD_MAIN(LOGICAL FLAG, INTEGER IERROR)
MPI_Query_thread	int MPI_Query_thread(int *provided);
MPI::Query_thread	int MPI::Query_thread();
MPI_QUERY_THREAD	MPI_QUERY_THREAD(INTEGER PROVIDED, INTEGER IERROR)
MPI_Wtick	double MPI_Wtick(void);
MPI::Wtick	double MPI::Wtick();
MPI_WTICK	DOUBLE PRECISION MPI_WTICK()
MPI_Wtime	double MPI_Wtime(void);
MPI::Wtime	double MPI::Wtime();
MPI_WTIME	DOUBLE PRECISION MPI_WTIME()

## Bindings for external interfaces

Table 31 lists the bindings for external interfaces.

Table 31. Binding for external interfaces

Subroutine name: C C++ FORTRAN	Binding: C C++ FORTRAN
MPI_Add_error_class	int MPI_Add_error_class(int *errorclass);
MPI::Add_error_class	int MPI::Add_error_class();

Table 31. Binding for external interfaces (continued)

Subroutine name: C C++ FORTRAN	Binding: C C++ FORTRAN
MPI_ADD_ERROR_CLASS	MPI_ADD_ERROR_CLASS(INTEGER ERRORCLASS, INTEGER IERROR)
MPI_Add_error_code	int MPI_Add_error_code(int errorclass, int *errorcode);
MPI::Add_error_code	int MPI::Add_error_code(int errorclass);
MPI_ADD_ERROR_CODE	MPI_ADD_ERROR_CODE(INTEGER ERRORCLASS, INTEGER ERRORCODE, INTEGER IERROR)
MPI_Add_error_string	int MPI_Add_error_string(int errorcode, char *string);
MPI::Add_error_string	void MPI::Add_error_string(int errorcode, const char* string);
MPI_ADD_ERROR_STRING	MPI_ADD_ERROR_STRING(INTEGER ERRORCODE, CHARACTER*(*) STRING, INTEGER IERROR)
MPI_Comm_call_errhandler	int MPI_Comm_call_errhandler (MPI_Comm comm, int errorcode);
MPI::Comm::Call_errhandler	void MPI::Comm::Call_errhandler(int errorcode) const;
MPI_COMM_CALL_ERRHANDLER	MPI_COMM_CALL_ERRHANDLER(INTEGER COMM, INTEGER ERRORCODE, INTEGER IERROR)
MPI_Comm_get_name	int MPI_Comm_get_name (MPI_Comm comm, char *comm_name, int *resultlen);
MPI::Comm::Get_name	void MPI::Comm::Get_name(char* comm_name, int& resultlen) const;
MPI_COMM_GET_NAME	MPI_COMM_GET_NAME(INTEGER COMM, CHARACTER*(*) COMM_NAME, INTEGER RESULTLEN, INTEGER IERROR)
MPI_Comm_set_name	int MPI_Comm_set_name (MPI_Comm comm, char *comm_name);
MPI::Comm::Set_name	void MPI::Comm::Set_name(const char* comm_name);
MPI_COMM_SET_NAME	MPI_COMM_SET_NAME(INTEGER COMM, CHARACTER*(*) COMM_NAME, INTEGER IERROR)
MPI_File_call_errhandler	int MPI_File_call_errhandler (MPI_File fh, int errorcode);
MPI::File::Call_errhandler	void MPI::File::Call_errhandler(int errorcode) const;
MPI_FILE_CALL_ERRHANDLER	MPI_FILE_CALL_ERRHANDLER(INTEGER FH, INTEGER ERRORCODE, INTEGER IERROR)
MPI_Grequest_complete	int MPI_Grequest_complete(MPI_Request request);
MPI::Grequest::Complete	void MPI::Grequest::Complete();
MPI_GREQUEST_COMPLETE	MPI_GREQUEST_COMPLETE(INTEGER REQUEST, INTEGER IERROR)
MPI_Grequest_start	int MPI_Grequest_start(MPI_Grequest_query_function *query_fn, MPI_Grequest_free_function *free_fn, MPI_Grequest_cancel_function *cancel_fn, void *extra_state, MPI_Request *request);
MPI::Grequest::Start	MPI::Grequest MPI::Grequest::Start(MPI::Grequest::Query_function query_fn, MPI::Grequest::Free_function free_fn, MPI::Grequest::Cancel_function cancel_fn, void *extra_state);
MPI_GREQUEST_START	MPI_GREQUEST_START(EXTERNAL QUERY_FN, EXTERNAL FREE_FN, EXTERNAL CANCEL_FN, INTEGER EXTRA_STATE, INTEGER REQUEST, INTEGER IERROR)
MPI_Status_set_elements	int MPI_Status_set_elements(MPI_Status *status, MPI_Datatype datatype, int count);

Table 31. Binding for external interfaces (continued)

Subroutine name: C C++ FORTRAN	Binding: C C++ FORTRAN
MPI::Status::Set_elements	void MPI::Status::Set_elements(const MPI::Datatype& datatype, int count);
MPI_STATUS_SET_CANCELLED	MPI_STATUS_SET_CANCELLED(INTEGER STATUS(MPI_STATUS_SIZE), LOGICAL FLAG, INTEGER IERROR)
MPI_Status_set_cancelled	int MPI_Status_set_cancelled(MPI_Status *status, int flag);
MPI::Status::Set_cancelled	void MPI::Status::Set_cancelled(bool flag);
MPI_STATUS_SET_ELEMENTS	MPI_STATUS_SET_ELEMENTS(INTEGER STATUS(MPI_STATUS_SIZE), INTEGER DATATYPE, INTEGER COUNT, INTEGER IERROR)
MPI_Type_get_name	int MPI_Type_get_name(MPI_Datatype type, char *type_name, int *resultlen);
MPI::Datatype::Get_name	void MPI::Datatype::Get_name(char* type_name, int& resultlen) const;
MPI_TYPE_GET_NAME	MPI_TYPE_GET_NAME(INTEGER TYPE, CHARACTER*(*) TYPE_NAME, INTEGER RESULTLEN, INTEGER IERROR)
MPI_Type_set_name	int MPI_Type_set_name (MPI_Datatype type, char *type_name);
MPI::Datatype::Set_name	void MPI::Datatype::Set_name(const char* type_name);
MPI_TYPE_SET_NAME	MPI_TYPE_SET_NAME(INTEGER TYPE, CHARACTER*(*) TYPE_NAME, INTEGER IERROR)
MPI_Win_call_errhandler	int MPI_Win_call_errhandler (MPI_Win win, int errorcode);
MPI::Win::Call_errhandler	void MPI::Win::Call_errhandler(int errorcode) const;
MPI_WIN_CALL_ERRHANDLER	MPI_WIN_CALL_ERRHANDLER(INTEGER WIN, INTEGER ERRORCODE, INTEGER IERROR)
MPI_Win_get_name	int MPI_Win_get_name (MPI_Win win, char *win_name, int *resultlen);
MPI::Win::Get_name	void MPI::Win::Get_name(char* win_name, int& resultlen) const;
MPI_WIN_GET_NAME	MPI_WIN_GET_NAME(INTEGER WIN, CHARACTER*(*) WIN_NAME, INTEGER RESULTLEN, INTEGER IERROR)
MPI_Win_set_name	int MPI_Win_set_name (MPI_Win win, char *win_name);
MPI::Win::Set_name	void MPI::Win::Set_name(const char* win_name);
MPI_WIN_SET_NAME	MPI_WIN_SET_NAME(INTEGER WIN, CHARACTER*(*) WIN_NAME, INTEGER IERROR)

## Bindings for group management

Table 32 lists the bindings for group management subroutines.

Table 32. Bindings for groups

Subroutine name: C C++ FORTRAN	Binding: C C++ FORTRAN
MPI_Comm_group	int MPI_Comm_group(MPI_Comm comm, MPI_Group *group);
MPI::Comm::Get_group	MPI::Group MPI::Comm::Get_group() const;

Table 32. Bindings for groups (continued)

Subroutine name: C C++ FORTRAN	Binding: C C++ FORTRAN
MPI_COMM_GROUP	MPI_COMM_GROUP(INTEGER COMM,INTEGER GROUP,INTEGER IERROR)
MPI_Group_compare	int MPI_Group_compare(MPI_Group group1,MPI_Group group2,int *result);
MPI::Group::Compare	static int MPI::Group::Compare(const MPI::Group& group1, const MPI::Group& group2);
MPI_GROUP_COMPARE	MPI_GROUP_COMPARE(INTEGER GROUP1,INTEGER GROUP2,INTEGER RESULT,INTEGER IERROR)
MPI_Group_difference	int MPI_Group_difference(MPI_Group group1,MPI_Group group2,MPI_Group *newgroup);
MPI::Group::Difference	static MPI::Group MPI::Group::Difference(const MPI::Group& group1, const MPI::Group& group2);
MPI_GROUP_DIFFERENCE	MPI_GROUP_DIFFERENCE(INTEGER GROUP1,INTEGER GROUP2,INTEGER NEWGROUP,INTEGER IERROR)
MPI_Group_excl	int MPI_Group_excl(MPI_Group group,int n,int *ranks,MPI_Group *newgroup);
MPI::Group::Excl	MPI::Group MPI::Group::Excl(int n, const int ranks[]) const;
MPI_GROUP_EXCL	MPI_GROUP_EXCL(INTEGER GROUP,INTEGER N,INTEGER RANKS(*),INTEGER NEWGROUP,INTEGER IERROR)
MPI_Group_free	int MPI_Group_free(MPI_Group *group);
MPI::Group::Free	void MPI::Group::Free();
MPI_GROUP_FREE	MPI_GROUP_FREE(INTEGER GROUP,INTEGER IERROR)
MPI_Group_incl	int MPI_Group_incl(MPI_Group group,int n,int *ranks,MPI_Group *newgroup);
MPI::Group::Incl	MPI::Group MPI::Group::Incl(int n, const int ranks[]) const;
MPI_GROUP_INCL	MPI_GROUP_INCL(INTEGER GROUP,INTEGER N,INTEGER RANKS(*),INTEGER NEWGROUP,INTEGER IERROR)
MPI_Group_intersection	int MPI_Group_intersection(MPI_Group group1,MPI_Group group2,MPI_Group *newgroup);
MPI::Group::Intersect	static MPI::Group MPI::Group::Intersect(const MPI::Group& group1, const MPI::Group& group2);
MPI_GROUP_INTERSECTION	MPI_GROUP_INTERSECTION(INTEGER GROUP1,INTEGER GROUP2,INTEGER NEWGROUP,INTEGER IERROR)
MPI_Group_range_excl	int MPI_Group_range_excl(MPI_Group group,int n,int ranges[][3],MPI_Group *newgroup);
MPI::Group::Range_excl	MPI::Group MPI::Group::Range_excl(int n, const int ranges[][3]) const;
MPI_GROUP_RANGE_EXCL	MPI_GROUP_RANGE_EXCL(INTEGER GROUP,INTEGER N,INTEGER RANGES(3,*),INTEGER NEWGROUP,INTEGER IERROR)
MPI_Group_range_incl	int MPI_Group_range_incl(MPI_Group group,int n,int ranges[][3],MPI_Group *newgroup);
MPI::Group::Range_incl	MPI::Group MPI::Group::Range_incl(int n, const int ranges[][3]) const;
MPI_GROUP_RANGE_INCL	MPI_GROUP_RANGE_INCL(INTEGER GROUP,INTEGER N,INTEGER RANGES(3,*),INTEGER NEWGROUP,INTEGER IERROR)

Table 32. Bindings for groups (continued)

Subroutine name: C C++ FORTRAN	Binding: C C++ FORTRAN
MPI_Group_rank	int MPI_Group_rank(MPI_Group group,int *rank);
MPI::Group::Get_rank	int MPI::Group::Get_rank() const;
MPI_GROUP_RANK	MPI_GROUP_RANK(INTEGER GROUP,INTEGER RANK,INTEGER IERROR)
MPI_Group_size	int MPI_Group_size(MPI_Group group,int *size);
MPI::Group::Get_size	int MPI::Group::Get_size() const;
MPI_GROUP_SIZE	MPI_GROUP_SIZE(INTEGER GROUP,INTEGER SIZE,INTEGER IERROR)
MPI_Group_translate_ranks	int MPI_Group_translate_ranks (MPI_Group group1,int n,int *ranks1,MPI_Group group2,int *ranks2);
MPI::Group::Translate_ranks	void MPI::Group::Translate_ranks(const MPI::Group& group1, int n, const int ranks1[], const MPI::Group& group2, int ranks2[]);
MPI_GROUP_TRANSLATE_RANKS	MPI_GROUP_TRANSLATE_RANKS(INTEGER GROUP1, INTEGER N,INTEGER RANKS1(*),INTEGER GROUP2,INTEGER RANKS2(*),INTEGER IERROR)
MPI_Group_union	int MPI_Group_union(MPI_Group group1,MPI_Group group2,MPI_Group *newgroup);
MPI::Group::Union	static MPI::Group MPI::Group::Union(const MPI::Group& group1, const MPI::Group& group2);
MPI_GROUP_UNION	MPI_GROUP_UNION(INTEGER GROUP1,INTEGER GROUP2,INTEGER NEWGROUP,INTEGER IERROR)

## Bindings for Info objects

Table 33 lists the bindings for Info objects.

Table 33. Bindings for Info objects

Subroutine name: C C++ FORTRAN	Binding: C C++ FORTRAN
MPI_Info_create	int MPI_Info_create(MPI_Info *info);
MPI::Info::Create	static MPI::Info MPI::Info::Create();
MPI_INFO_CREATE	MPI_INFO_CREATE(INTEGER INFO,INTEGER IERROR)
MPI_Info_delete	int MPI_Info_delete(MPI_Info info,char *key);
MPI::Info::Delete	void MPI::Info::Delete(const char* key);
MPI_INFO_DELETE	MPI_INFO_DELETE(INTEGER INFO,CHARACTER KEY(*), INTEGER IERROR)
MPI_Info_dup	int MPI_Info_dup(MPI_Info info,MPI_Info *newinfo);
MPI::Info::Dup	MPI::Info MPI::Info::Dup() const;
MPI_INFO_DUP	MPI_INFO_DUP(INTEGER INFO,INTEGER NEWINFO,INTEGER IERROR)
MPI_Info_free	int MPI_Info_free(MPI_Info *info);

Table 33. Bindings for Info objects (continued)

Subroutine name: C C++ FORTRAN	Binding: C C++ FORTRAN
MPI::Info::Free	void MPI::Info::Free();
MPI_INFO_FREE	MPI_INFO_FREE(INTEGER INFO,INTEGER IERROR)
MPI_Info_get	int MPI_Info_get(MPI_Info info,char *key,int valuelen, char *value,int *flag);
MPI::Info::Get	bool MPI::Info::Get(const char* key, int valuelen, char* value) const;
MPI_INFO_GET	MPI_INFO_GET (INTEGER INFO,CHARACTER KEY(*),INTEGER VALUELEN, CHARACTER VALUE(*),LOGICAL FLAG,INTEGER IERROR)
MPI_Info_get_nkeys	int MPI_Info_get_nkeys(MPI_Info info,int *nkeys);
MPI::Info::Get_nkeys	int MPI::Info::Get_nkeys() const;
MPI_INFO_GET_NKEYS	MPI_INFO_GET_NKEYS(INTEGER INFO,INTEGER NKEYS,INTEGER IERROR)
MPI_Info_get_nthkey	int MPI_Info_get_nthkey(MPI_Info info, int n, char *key);
MPI::Info::Get_nthkey	void MPI::Info::Get_nthkey(int n, char* key) const;
MPI_INFO_GET_NTHKEY	MPI_INFO_GET_NTHKEY(INTEGER INFO,INTEGER N,CHARACTER KEY(*), INTEGER IERROR)
MPI_Info_get_valuelen	int MPI_Info_get_valuelen(MPI_Info info,char *key,int *valuelen, int *flag);
MPI::Info::Get_valuelen	bool MPI::Info::Get_valuelen(const char* key, int& valuelen) const;
MPI_INFO_GET_VALUELEN	MPI_INFO_GET_VALUELEN(INTEGER INFO,CHARACTER KEY(*),INTEGER VALUELEN,LOGICAL FLAG, INTEGER IERROR)
MPI_Info_set	int MPI_Info_set(MPI_Info info,char *key,char *value);
MPI::Info::Set	void MPI::Info::Set(const char* key, const char* value);
MPI_INFO_SET	MPI_INFO_SET(INTEGER INFO,CHARACTER KEY(*),CHARACTER VALUE(*), INTEGER IERROR)

## Bindings for memory allocation

Table 34 lists the bindings for memory allocation subroutines.

Table 34. Bindings for memory allocation

Subroutine name: C C++ FORTRAN	Binding: C C++ FORTRAN
MPI_Alloc_mem	int MPI_Alloc_mem (MPI_Aint size, MPI_Info info, void *baseptr);
MPI::Alloc_mem	void* MPI::Alloc_mem(MPI::Aint size, const MPI::Info& info);
MPI_ALLOC_MEM	MPI_ALLOC_MEM(INTEGER SIZE, INTEGER INFO, INTEGER BASEPTR, INTEGER IERROR)
MPI_Free_mem	int MPI_Free_mem (void *base);
MPI::Free_mem	void MPI::Free_mem(void *base);
MPI_FREE_MEM	MPI_FREE_MEM(CHOICE BASE, INTEGER IERROR)

## Bindings for MPI-IO

Table 35 lists the bindings for MPI-IO subroutines.

Table 35. Bindings for MPI-IO

Subroutine name: C C++ FORTRAN	Binding: C C++ FORTRAN
MPI_File_close	int MPI_File_close (MPI_File *fh);
MPI::File::Close	void MPI::File::Close();
MPI_FILE_CLOSE	MPI_FILE_CLOSE(INTEGER FH,INTEGER IERROR)
MPI_File_delete	int MPI_File_delete (char *filename,MPI_Info info);
MPI::File::Delete	static void MPI::File::Delete(const char* filename, const MPI::Info& info);
MPI_FILE_DELETE	MPI_FILE_DELETE(CHARACTER*(*) FILENAME,INTEGER INFO,INTEGER IERROR)
MPI_File_get_amode	int MPI_File_get_amode (MPI_File fh,int *amode);
MPI::File::Get_amode	int MPI::File::Get_amode() const;
MPI_FILE_GET_AMODE	MPI_FILE_GET_AMODE(INTEGER FH,INTEGER AMODE,INTEGER IERROR)
MPI_File_get_atomicity	int MPI_File_get_atomicity (MPI_File fh,int *flag);
MPI::File::Get_atomicity	bool MPI::File::Get_atomicity() const;
MPI_FILE_GET_ATOMICITY	MPI_FILE_GET_ATOMICITY (INTEGER FH,LOGICAL FLAG,INTEGER IERROR)
MPI_File_get_byte_offset	int MPI_File_get_byte_offset(MPI_File fh, MPI_Offset offset, MPI_Offset *disp);
MPI::File::Get_byte_offset	MPI::Offset MPI::File::Get_byte_offset(const MPI::Offset disp) const;
MPI_FILE_GET_BYTE_OFFSET	MPI_FILE_GET_BYTE_OFFSET(INTEGER FH,INTEGER(KIND=MPI_OFFSET_KIND) OFFSET,INTEGER(KIND=MPI_OFFSET_KIND) DISP,INTEGER IERROR)
MPI_File_get_group	int MPI_File_get_group (MPI_File fh,MPI_Group *group);
MPI::File::Get_group	MPI::Group MPI::File::Get_group() const;
MPI_FILE_GET_GROUP	MPI_FILE_GET_GROUP (INTEGER FH,INTEGER GROUP,INTEGER IERROR)
MPI_File_get_info	int MPI_File_get_info (MPI_File fh,MPI_Info *info_used);
MPI::File::Get_info	MPI::Info MPI::File::Get_info() const;
MPI_FILE_GET_INFO	MPI_FILE_GET_INFO (INTEGER FH,INTEGER INFO_USED,INTEGER IERROR)
MPI_File_get_position	int MPI_File_get_position(MPI_File fh,MPI_Offset *offset);
MPI::File::Get_position	MPI::Offset MPI::File::Get_position() const;
MPI_FILE_GET_POSITION	MPI_FILE_GET_POSITION(INTEGER FH,INTEGER(KIND=MPI_OFFSET_KIND) OFFSET,INTEGER IERROR)
MPI_File_get_position_shared	int MPI_File_get_position_shared(MPI_File fh, MPI_Offset *offset);
MPI::File::Get_position_shared	MPI::Offset MPI::File::Get_position_shared() const;
MPI_FILE_GET_POSITION_SHARED	MPI_FILE_GET_POSITION_SHARED(INTEGER FH,INTEGER(KIND=MPI_OFFSET_KIND) OFFSET,INTEGER IERROR)
MPI_File_get_size	int MPI_File_get_size (MPI_File fh,MPI_Offset size);

Table 35. Bindings for MPI-IO (continued)

Subroutine name: C C++ FORTRAN	Binding: C C++ FORTRAN
MPI::File::Get_size	MPI::Offset MPI::File::Get_size() const;
MPI_FILE_GET_SIZE	MPI_FILE_GET_SIZE (INTEGER FH,INTEGER(KIND=MPI_OFFSET_KIND) SIZE, INTEGER IERROR)
MPI_File_get_type_extent	int MPI_File_get_type_extent(MPI_File fh, MPI_Datatype datatype, MPI_Aint *extent);
MPI::File::Get_type_extent	MPI::Aint MPI::File::Get_type_extent(const MPI::Datatype& datatype) const;
MPI_FILE_GET_TYPE_EXTENT	MPI_FILE_GET_TYPE_EXTENT (INTEGER FH, INTEGER DATATYPE, INTEGER(KIND=MPI_ADDRESS_KIND) EXTENT, INTEGER IERROR)
MPI_File_get_view	int MPI_File_get_view (MPI_File fh,MPI_Offset *disp, MPI_Datatype *etype,MPI_Datatype *filetype,char *datarep);
MPI::File::Get_view	void MPI::File::Get_view(MPI::Offset& disp,MPI::Datatype& etype, MPI::Datatype& filetype, char* datarep) const;
MPI_FILE_GET_VIEW	MPI_FILE_GET_VIEW (INTEGER FH,INTEGER(KIND=MPI_OFFSET_KIND) DISP, INTEGER ETYPE,INTEGER FILETYPE,INTEGER DATAREP,INTEGER IERROR)
MPI_File_iread	int MPI_File_iread (MPI_File fh,void *buf, int count, MPI_Datatype datatype,MPI_Request *request);
MPI::File::Iread	MPI::Request MPI::File::Iread(void* buf, int count, const MPI::Datatype& datatype);
MPI_FILE_IREAD	MPI_FILE_IREAD (INTEGER FH, CHOICE BUF, INTEGER COUNT, INTEGER DATATYPE, INTEGER REQUEST, INTEGER IERROR)
MPI_File_iread_at	int MPI_File_iread_at (MPI_File fh,MPI_Offset offset,void *buf, int count,MPI_Datatype datatype,MPI_Request *request);
MPI::File::Iread_at	MPI::Request MPI::File::Iread_at(MPI::Offset offset, void* buf, int count, const MPI::Datatype& datatype);
MPI_FILE_IREAD_AT	MPI_FILE_IREAD_AT (INTEGER FH,INTEGER (KIND=MPI_OFFSET_KIND) OFFSET, CHOICE BUF,INTEGER COUNT,INTEGER DATATYPE,INTEGER REQUEST, INTEGER IERROR)
MPI_File_iread_shared	int MPI_File_iread_shared (MPI_File fh,void *buf, int count, MPI_Datatype datatype,MPI_Request *request);
MPI::File::Iread_shared	MPI::Request MPI::File::Iread_shared(void* buf, int count, const MPI::Datatype& datatype);
MPI_FILE_IREAD_SHARED	MPI_FILE_IREAD_SHARED (INTEGER FH, CHOICE BUF, INTEGER COUNT, INTEGER DATATYPE, INTEGER REQUEST, INTEGER IERROR)
MPI_File_fwrite	int MPI_File_fwrite (MPI_File fh, void *buf, int count, MPI_Datatype datatype,MPI_Request *request);
MPI::File::Iwrite	MPI::Request MPI::File::Iwrite(const void* buf, int count, const MPI::Datatype& datatype);
MPI_FILE_IWRITE	MPI_FILE_IWRITE(INTEGER FH,CHOICE BUF,INTEGER COUNT,INTEGER DATATYPE, INTEGER REQUEST,INTEGER IERROR)

Table 35. Bindings for MPI-IO (continued)

Subroutine name: C C++ FORTRAN	Binding: C C++ FORTRAN
MPI_File_iread_at	int MPI_File_iread_at (MPI_File fh, MPI_Offset offset, void *buf, int count, MPI_Datatype datatype, MPI_Request *request);
MPI::File::Iread_at	MPI::Request MPI::File::Iread_at(MPI::Offset offset, const void* buf, int count, const MPI::Datatype& datatype);
MPI_FILE_IWRITE_AT	MPI_FILE_IWRITE_AT(INTEGER FH, INTEGER(KIND=MPI_OFFSET_KIND) OFFSET, CHOICE BUF, INTEGER COUNT, INTEGER DATATYPE, INTEGER REQUEST, INTEGER IERROR)
MPI_File_iread_shared	int MPI_File_iread_shared (MPI_File fh, void *buf, int count, MPI_Datatype datatype, MPI_Request *request);
MPI::File::Iread_shared	MPI::Request MPI::File::Iread_shared(const void* buf, int count, const MPI::Datatype& datatype);
MPI_FILE_IWRITE_SHARED	MPI_FILE_IWRITE_SHARED (INTEGER FH, CHOICE BUF, INTEGER COUNT, INTEGER DATATYPE, INTEGER REQUEST, INTEGER IERROR)
MPI_File_open	int MPI_File_open (MPI_Comm comm, char *filename, int amode, MPI_info, MPI_File *fh);
MPI::File::Open	static MPI::File MPI::File::Open(const MPI::Intracomm& comm, const char* filename, int amode, const MPI::Info& info);
MPI_FILE_OPEN	MPI_FILE_OPEN(INTEGER COMM, CHARACTER FILENAME(*), INTEGER AMODE, INTEGER INFO, INTEGER FH, INTEGER IERROR)
MPI_File_preallocate	int MPI_File_preallocate (MPI_File fh, MPI_Offset size);
MPI::File::Preallocate	void MPI::File::Preallocate(MPI::Offset size);
MPI_FILE_PREALLOCATE	MPI_FILE_PREALLOCATE(INTEGER FH, INTEGER SIZE, INTEGER IERROR)
MPI_File_read	int MPI_File_read (MPI_File fh, void *buf, int count, MPI_Datatype datatype, MPI_Status *status);
MPI::File::Read	void MPI::File::Read(void* buf, int count, const MPI::Datatype& datatype, MPI::Status& status);
MPI_FILE_READ	MPI_FILE_READ(INTEGER FH, CHOICE BUF, INTEGER COUNT, INTEGER DATATYPE, INTEGER STATUS(MPI_STATUS_SIZE), INTEGER IERROR)
MPI_File_read_all	int MPI_File_read_all (MPI_File fh, void *buf, int count, MPI_Datatype datatype, MPI_Status *status);
MPI::File::Read_all	void MPI::File::Read_all(void* buf, int count, const MPI::Datatype& datatype, MPI::Status& status);  void MPI::File::Read_all(void* buf, int count, const MPI::Datatype& datatype);
MPI_FILE_READ_ALL	MPI_FILE_READ_ALL(INTEGER FH, CHOICE BUF, INTEGER COUNT, INTEGER DATATYPE, INTEGER STATUS(MPI_STATUS_SIZE), INTEGER IERROR)
MPI_File_read_all_begin	int MPI_File_read_all_begin (MPI_File fh, void *buf, int count, MPI_Datatype datatype);

Table 35. Bindings for MPI-IO (continued)

Subroutine name: C C++ FORTRAN	Binding: C C++ FORTRAN
MPI::File::Read_all_begin	void MPI::File::Read_all_begin(void* buf, int count, const MPI::Datatype& datatype);
MPI_FILE_READ_ALL_BEGIN	MPI_FILE_READ_ALL_BEGIN (INTEGER FH, CHOICE BUF, INTEGER COUNT, INTEGER DATATYPE, INTEGER IERROR)
MPI_File_read_all_end	int MPI_File_read_all_end(MPI_File fh, void* buf, MPI_Status* status);
MPI::File::Read_all_end	void MPI::File::Read_all_end(void* buf);  void MPI::File::Read_all_end(void* buf, MPI::Status& status);
MPI_FILE_READ_ALL_END	MPI_FILE_READ_ALL_END(INTEGER FH, CHOICE BUF, INTEGER STATUS(MPI_STATUS_SIZE), INTEGER IERROR)
MPI_File_read_at	int MPI_File_read_at (MPI_File fh, MPI_Offset offset, void* buf, int count, MPI_Datatype datatype, MPI_Status* status);
MPI::File::Read_at	void MPI::File::Read_at(MPI::Offset offset, void* buf, int count, const MPI::Datatype& datatype);  void MPI::File::Read_at(MPI::Offset offset, void* buf, int count, const MPI::Datatype& datatype, MPI::Status& status);
MPI_FILE_READ_AT	MPI_FILE_READ_AT(INTEGER FH, INTEGER(KIND=MPI_OFFSET_KIND) OFFSET, CHOICE BUF, INTEGER COUNT, INTEGER DATATYPE, INTEGER STATUS(MPI_STATUS_SIZE), INTEGER IERROR)
MPI_File_read_at_all	int MPI_File_read_at_all (MPI_File fh, MPI_Offset offset, void* buf, int count, MPI_Datatype datatype, MPI_Status* status);
MPI::File::Read_at_all	void MPI::File::Read_at_all(MPI::Offset offset, void* buf, int count, const MPI::Datatype& datatype);  void MPI::File::Read_at_all(MPI::Offset offset, void* buf, int count, const MPI::Datatype& datatype, MPI::Status& status);
MPI_FILE_READ_AT_ALL	MPI_FILE_READ_AT_ALL(INTEGER FH, INTEGER(KIND=MPI_OFFSET_KIND) OFFSET, CHOICE BUF, INTEGER COUNT, INTEGER DATATYPE, INTEGER STATUS(MPI_STATUS_SIZE), INTEGER IERROR)
MPI_File_read_at_all_begin	int MPI_File_read_at_all_begin(MPI_File fh, MPI_Offset offset, void* buf, int count, MPI_Datatype datatype);
MPI::File::Read_at_all_begin	void MPI::File::Read_at_all_begin(MPI::Offset offset, void* buf, int count, const MPI::Datatype& datatype);
MPI_FILE_READ_AT_ALL_BEGIN	MPI_FILE_READ_AT_ALL_BEGIN(INTEGER FH, INTEGER(KIND=MPI_OFFSET_KIND) OFFSET, CHOICE BUF, INTEGER COUNT, INTEGER DATATYPE, INTEGER IERROR)
MPI_File_read_at_all_end	int MPI_File_read_at_all_end(MPI_File fh, void* buf, MPI_Status* status);
MPI::File::Read_at_all_end	void MPI::File::Read_at_all_end(void* buf, MPI::Status& status);  void MPI::File::Read_at_all_end(void* buf);
MPI_FILE_READ_AT_ALL_END	MPI_FILE_READ_AT_ALL_END(INTEGER FH, CHOICE BUF, INTEGER STATUS(MPI_STATUS_SIZE), INTEGER IERROR)
MPI_File_read_ordered	int MPI_File_read_ordered(MPI_File fh, void* buf, int count, MPI_Datatype datatype, MPI_Status* status);

Table 35. Bindings for MPI-IO (continued)

Subroutine name: C C++ FORTRAN	Binding: C C++ FORTRAN
MPI::File::Read_ordered	void MPI::File::Read_ordered(void* buf, int count, const MPI::Datatype& datatype, MPI::Status& status);
MPI_FILE_READ_ORDERED	MPI_FILE_READ_ORDERED(INTEGER FH,CHOICE BUF,INTEGER COUNT, INTEGER DATATYPE, INTEGER STATUS(MPI_STATUS_SIZE),INTEGER IERROR)
MPI_File_read_ordered_begin	int MPI_File_read_ordered_begin(MPI_File fh, void *buf, int count, MPI_Datatype datatype);
MPI::File::Read_ordered_begin	void MPI::File::Read_ordered_begin(void* buf, int count, const MPI::Datatype& datatype);
MPI_FILE_READ_ORDERED_BEGIN	MPI_FILE_READ_ORDERED_BEGIN (INTEGER FH, CHOICE BUF, INTEGER COUNT, INTEGER DATATYPE, INTEGER IERROR)
MPI_File_read_ordered_end	int MPI_File_read_ordered_end(MPI_File fh,void *buf, MPI_Status *status)
MPI::File::Read_ordered_end	void MPI::File::Read_ordered_end(void* buf, MPI::Status& status);  void MPI::File::Read_ordered_end(void* buf);
MPI_FILE_READ_ORDERED_END	MPI_FILE_READ_ORDERED_END(INTEGER FH,CHOICE BUF, INTEGER STATUS(MPI_STATUS_SIZE), INTEGER IERROR)
MPI_File_read_shared	int MPI_File_read_shared (MPI_File fh, void *buf, int count, MPI_Datatype datatype,MPI_Status *status);
MPI::File::Read_shared	void MPI::File::Read_shared(void* buf, int count, const MPI::Datatype& datatype);  void MPI::File::Read_shared(void* buf, int count, const MPI::Datatype& datatype, MPI::Status& status);
MPI_FILE_READ_SHARED	MPI_FILE_READ_SHARED(INTEGER FH,CHOICE BUF,INTEGER COUNT, INTEGER DATATYPE, INTEGER STATUS(MPI_STATUS_SIZE),INTEGER IERROR)
MPI_File_seek	int MPI_File_seek (MPI_File fh,MPI_Offset offset, int whence);
MPI::File::Seek	void MPI::File::Seek(MPI::Offset offset, int whence);
MPI_FILE_SEEK	MPI_FILE_SEEK (INTEGER FH, INTEGER(KIND=MPI_OFFSET_KIND) OFFSET, INTEGER WHENCE, INTEGER IERROR)
MPI_File_seek_shared	int MPI_File_seek_shared(MPI_File fh,MPI_Offset offset, int whence);
MPI::File::Seek_shared	void MPI::File::Seek_shared(MPI::Offset offset, int whence);
MPI_FILE_SEEK_SHARED	MPI_FILE_SEEK_SHARED(INTEGER FH,INTEGER(KIND=MPI_OFFSET_KIND) OFFSET, INTEGER WHENCE, INTEGER IERROR)
MPI_File_set_atomicsity	int MPI_File_set_atomicsity (MPI_File fh,int flag);
MPI::File::Set_atomicsity	void MPI::File::Set_atomicsity(bool flag);
MPI_FILE_SET_ATOMICITY	MPI_FILE_SET_ATOMICITY (INTEGER FH,LOGICAL FLAG,INTEGER IERROR)
MPI_File_set_info	int MPI_File_set_info (MPI_File fh,MPI_Info info);
MPI::File::Set_info	void MPI::File::Set_info(const MPI::Info& info);

Table 35. Bindings for MPI-IO (continued)

Subroutine name: C C++ FORTRAN	Binding: C C++ FORTRAN
MPI_FILE_SET_INFO	MPI_FILE_SET_INFO(INTEGER FH,INTEGER INFO,INTEGER IERROR)
MPI_File_set_size	int MPI_File_set_size (MPI_File fh,MPI_Offset size);
MPI::File::Set_size	void MPI::File::Set_size(MPI::Offset size);
MPI_FILE_SET_SIZE	MPI_FILE_SET_SIZE (INTEGER FH,INTEGER(KIND=MPI_OFFSET_KIND) SIZE, INTEGER IERROR)
MPI_File_set_view	int MPI_File_set_view (MPI_File fh,MPI_Offset disp, MPI_Datatype etype,MPI_Datatype filetype, char *datarep,MPI_Info info);
MPI::File::Set_view	void MPI::File::Set_view(MPI::Offset disp, const MPI::Datatype& etype, const MPI::Datatype& filetype, const char* datarep, const MPI::Info& info);
MPI_FILE_SET_VIEW	MPI_FILE_SET_VIEW (INTEGER FH,INTEGER(KIND=MPI_OFFSET_KIND) DISP, INTEGER ETYPE,INTEGER FILETYPE,CHARACTER DATAREP(*),INTEGER INFO, INTEGER IERROR)
MPI_File_sync	int MPI_File_sync (MPI_File fh);
MPI::File::Sync	void MPI::File::Sync();
MPI_FILE_SYNC	MPI_FILE_SYNC (INTEGER FH,INTEGER IERROR)
MPI_File_write	int MPI_File_write (MPI_File fh, void *buf, int count, MPI_Datatype datatype,MPI_Status *status);
MPI::File::Write	void MPI::File::Write(const void* buf, int count, const MPI::Datatype& datatype);  void MPI::File::Write(const void* buf, int count, const MPI::Datatype& datatype, MPI::Status& status);
MPI_FILE_WRITE	MPI_FILE_WRITE(INTEGER FH,CHOICE BUF,INTEGER COUNT,INTEGER DATATYPE, INTEGER STATUS(MPI_STATUS_SIZE),INTEGER IERROR)
MPI_File_write_all	int MPI_File_write_all (MPI_File fh, void *buf, int count, MPI_Datatype datatype,MPI_Status *status);
MPI::File::Write_all	void MPI::File::Write_all(const void* buf, int count, const MPI::Datatype& datatype);  void MPI::File::Write_all(const void* buf, int count, const MPI::Datatype& datatype, MPI::Status& status);
MPI_FILE_WRITE_ALL	MPI_FILE_WRITE_ALL(INTEGER FH,CHOICE BUF,INTEGER COUNT, INTEGER DATATYPE,INTEGER STATUS(MPI_STATUS_SIZE),INTEGER IERROR)
MPI_File_write_all_begin	int MPI_File_write_all_begin (MPI_File fh, void *buf, int count, MPI_Datatype datatype);
MPI::File::Write_all_begin	void MPI::File::Write_all_begin(const void* buf, int count, const MPI::Datatype& datatype);
MPI_FILE_WRITE_ALL_BEGIN	MPI_FILE_WRITE_ALL_BEGIN (INTEGER FH, CHOICE BUF, INTEGER COUNT, INTEGER DATATYPE, INTEGER IERROR)
MPI_File_write_all_end	int MPI_File_write_all_end(MPI_File fh,void *buf, MPI_Status *status);

Table 35. Bindings for MPI-IO (continued)

Subroutine name: C C++ FORTRAN	Binding: C C++ FORTRAN
MPI::File::Write_all_end	void MPI::File::Write_all_end(void* buf);  void MPI::File::Write_all_end(void* buf, MPI::Status& status);
MPI_FILE_WRITE_ALL_END	MPI_FILE_WRITE_ALL_END(INTEGER FH, CHOICE BUF, INTEGER STATUS(MPI_STATUS_SIZE), INTEGER IERROR)
MPI_File_write_at	int MPI_File_write_at (MPI_File fh, MPI_Offset offset, void *buf, int count, MPI_Datatype datatype, MPI_Status *status);
MPI::File::Write_at	void MPI::File::Write_at(MPI::Offset offset, const void* buf, int count, const MPI::Datatype& datatype);  void MPI::File::Write_at(MPI::Offset offset, const void* buf, int count, const MPI::Datatype& datatype, MPI::Status& status);
MPI_FILE_WRITE_AT	MPI_FILE_WRITE_AT(INTEGER FH, INTEGER(KIND=MPI_OFFSET_KIND) OFFSET, CHOICE BUF, INTEGER COUNT, INTEGER DATATYPE, INTEGER STATUS(MPI_STATUS_SIZE), INTEGER IERROR)
MPI_File_write_at_all	int MPI_File_write_at_all (MPI_File fh, MPI_Offset offset, void *buf, int count, MPI_Datatype datatype, MPI_Status *status);
MPI::File::Write_at_all	void MPI::File::Write_at_all(MPI::Offset offset, const void* buf, int count, const MPI::Datatype& datatype);  void MPI::File::Write_at_all(MPI::Offset offset, const void* buf, int count, const MPI::Datatype& datatype, MPI::Status& status);
MPI_FILE_WRITE_AT_ALL	MPI_FILE_WRITE_AT_ALL (INTEGER FH, INTEGER (KIND=MPI_OFFSET_KIND) OFFSET, CHOICE BUF, INTEGER COUNT, INTEGER DATATYPE, INTEGER STATUS(MPI_STATUS_SIZE), INTEGER IERROR)
MPI_File_write_at_all_begin	int MPI_File_write_at_all_begin(MPI_File fh, MPI_Offset offset, void *buf, int count, MPI_Datatype datatype);
MPI::File::Write_at_all_begin	void MPI::File::Write_at_all_begin(MPI::Offset offset, const void* buf, int count, const MPI::Datatype& datatype);
MPI_FILE_WRITE_AT_ALL_BEGIN	MPI_FILE_WRITE_AT_ALL_BEGIN(INTEGER FH, INTEGER(KIND=MPI_OFFSET_KIND) OFFSET, CHOICE BUF, INTEGER COUNT, INTEGER DATATYPE, INTEGER IERROR)
MPI_File_write_at_all_end	int MPI_File_write_at_all_end(MPI_File fh, void *buf, MPI_Status *status);
MPI::File::Write_at_all_end	void MPI::File::Write_at_all_end(const void* buf);  void MPI::File::Write_at_all_end(const void* buf, MPI::Status& status);
MPI_FILE_WRITE_AT_ALL_END	MPI_FILE_WRITE_AT_ALL_END(INTEGER FH, CHOICE BUF, INTEGER STATUS(MPI_STATUS_SIZE), INTEGER IERROR)
MPI_File_write_ordered	int MPI_File_write_ordered(MPI_File fh, void *buf, int count, MPI_Datatype datatype, MPI_Status *status);
MPI::File::Write_ordered	void MPI::File::Write_ordered(const void* buf, int count, const MPI::Datatype& datatype);  void MPI::File::Write_ordered(const void* buf, int count, const MPI::Datatype& datatype, MPI::Status& status);

Table 35. Bindings for MPI-IO (continued)

Subroutine name: C C++ FORTRAN	Binding: C C++ FORTRAN
MPI_FILE_WRITE_ORDERED	MPI_FILE_WRITE_ORDERED(INTEGER FH,CHOICE BUF,INTEGER COUNT, INTEGER DATATYPE, INTEGER STATUS(MPI_STATUS_SIZE),INTEGER IERROR)
MPI_File_write_ordered_begin	int MPI_File_write_ordered_begin(MPI_File fh, void *buf, int count, MPI_Datatype datatype);
MPI::File::Write_ordered_begin	void MPI::File::Write_ordered_begin(const void* buf, int count, const MPI::Datatype& datatype);
MPI_FILE_WRITE_ORDERED_BEGIN	MPI_FILE_WRITE_ORDERED_BEGIN (INTEGER FH, CHOICE BUF, INTEGER COUNT, INTEGER DATATYPE, INTEGER IERROR)
MPI_File_write_ordered_end	int MPI_File_write_ordered_end(MPI_File fh,void *buf, MPI_Status *status)
MPI::File::Write_ordered_end	void MPI::File::Write_ordered_end(const void* buf);  void MPI::File::Write_ordered_end(const void* buf, MPI::Status& status);
MPI_FILE_WRITE_ORDERED_END	MPI_FILE_WRITE_ORDERED_END(INTEGER FH,CHOICE BUF, INTEGER STATUS(MPI_STATUS_SIZE), INTEGER IERROR)
MPI_File_write_shared	int MPI_File_write_shared (MPI_File fh, void *buf, int count, MPI_Datatype datatype,MPI_Status *status);
MPI::File::Write_shared	void MPI::File::Write_shared(const void* buf, int count, const MPI::Datatype& datatype);  void MPI::File::Write_shared(const void* buf, int count, const MPI::Datatype& datatype, MPI::Status& status);
MPI_FILE_WRITE_SHARED	MPI_FILE_WRITE_SHARED(INTEGER FH,CHOICE BUF,INTEGER COUNT, INTEGER DATATYPE, INTEGER STATUS(MPI_STATUS_SIZE),INTEGER IERROR)
MPI_Register_datarep	int MPI_Register_datarep(char *datarep, MPI_Datarep_conversion_function *read_conversion_fn, MPI_Datarep_conversion_function *write_conversion_fn, MPI_Datarep_extents_function *dtype_file_extents_fn, void *extra_state);
MPI::Register_datarep	void MPI::Register_datarep(const char* datarep, MPI::Datarep_conversion_function* read_conversion_fn, MPI::Datarep_conversion_function* write_conversion_fn, MPI::Datarep_extents_function* dtype_file_extents_fn, void* extra_state);
MPI_REGISTER_DATAREP	MPI_REGISTER_DATAREP(CHARACTER(*) DATAREP, EXTERNAL READ_CONVERSION_FN, EXTERNAL WRITE_CONVERSION_FN, EXTERNAL DTYPE_FILE_EXTENT_FN, INTEGER(KIND=MPI_ADDRESS_KIND), INTEGER EXTRA_STATE, INTEGER IERROR)

## Bindings for MPI\_Status objects

Table 36 on page 205 lists the bindings for MPI\_Status object subroutines.

Table 36. Bindings for MPI\_Status objects

Subroutine name: C C++ FORTRAN	Binding: C C++ FORTRAN
MPI_Request_get_status	int MPI_Request_get_status(MPI_Request request, int *flag, MPI_Status *status);
MPI::Request::Get_status	bool MPI::Request::Get_status() const;  bool MPI::Request::Get_status(MPI::Status&status) const;
MPI_REQUEST_GET_STATUS	MPI_REQUEST_GET_STATUS(INTEGER REQUEST, LOGICAL FLAG, INTEGER STATUS, INTEGER IERROR)

## Bindings for one-sided communication

Table 37 lists the bindings for one-sided communication subroutines.

Table 37. Bindings for one-sided communication

Subroutine name: C C++ FORTRAN	Binding: C C++ FORTRAN
MPI_Accumulate	int MPI_Accumulate (void *origin_addr, int origin_count, MPI_Datatype origin_datatype, int target_rank, MPI_Aint target_disp, int target_count, MPI_Datatype target_datatype, MPI_Op op, MPI_Win win);
MPI::Win::Accumulate	void MPI::Win::Accumulate(const void* origin_addr, int origin_count, const MPI::Datatype& origin_datatype, int target_rank, MPI::Aint target_disp, int target_count, const MPI::Datatype& target_datatype, const MPI::Op& op) const;
MPI_ACCUMULATE	MPI_ACCUMULATE (CHOICE ORIGIN_ADDR, INTEGER ORIGIN_COUNT, INTEGER ORIGIN_DATATYPE, INTEGER TARGET_RANK, INTEGER TARGET_DISP, INTEGER TARGET_COUNT, INTEGER TARGET_DATATYPE, INTEGER OP, INTEGER WIN, INTEGER IERROR)
MPI_Get	int MPI_Get (void *origin_addr, int origin_count, MPI_Datatype origin_datatype, int target_rank, MPI_Aint target_disp, int target_count, MPI_Datatype target_datatype, MPI_Win win);
MPI::Win::Get	void MPI::Win::Get(void* origin_addr, int origin_count, const MPI::Datatype& origin_datatype, int target_rank, MPI::Aint target_disp, int target_count, const MPI::Datatype& target_datatype) const;
MPI_GET	MPI_GET(CHOICE ORIGIN_ADDR, INTEGER ORIGIN_COUNT, INTEGER ORIGIN_DATATYPE, INTEGER TARGET_RANK, INTEGER TARGET_DISP, INTEGER TARGET_COUNT, INTEGER TARGET_DATATYPE, INTEGER WIN, INTEGER IERROR)
MPI_Put	int MPI_Put (void *origin_addr, int origin_count, MPI_Datatype origin_datatype, int target_rank, MPI_Aint target_disp, int target_count, MPI_Datatype target_datatype, MPI_Win win);
MPI::Win::Put	void MPI::Win::Put(const void* origin_addr, int origin_count, const MPI::Datatype& origin_datatype, int target_rank, MPI::Aint target_disp, int target_count, const MPI::Datatype& target_datatype) const;

Table 37. Bindings for one-sided communication (continued)

Subroutine name: C C++ FORTRAN	Binding: C C++ FORTRAN
MPI_PUT	MPI_PUT(CHOICE ORIGIN_ADDR, INTEGER ORIGIN_COUNT, INTEGER ORIGIN_DATATYPE, INTEGER TARGET_RANK, INTEGER TARGET_DISP, INTEGER TARGET_COUNT, INTEGER TARGET_DATATYPE, INTEGER WIN, INTEGER IERROR)
MPI_Win_complete	int MPI_Win_complete (MPI_Win win);
MPI::Win::Complete	void MPI::Win::Complete() const;
MPI_WIN_COMPLETE	MPI_WIN_COMPLETE(INTEGER WIN, INTEGER IERROR)
MPI_Win_create	int MPI_Win_create (void *base, MPI_Aint size, int disp_unit, MPI_Info info, MPI_Comm comm, MPI_Win *win); MPI_WIN_CREATE(CHOICE BASE, INTEGER SIZE, INTEGER DISP_UNIT, INTEGER INFO, INTEGER COMM, INTEGER WIN, INTEGER IERROR)
MPI::Win::Create	static MPI::Win MPI::Win::Create(const void* base, MPI::Aint size, int disp_unit, const MPI::Info& info, const MPI::Intracomm& comm);
MPI_WIN_CREATE	MPI_WIN_CREATE(CHOICE BASE, INTEGER SIZE, INTEGER DISP_UNIT, INTEGER INFO, INTEGER COMM, INTEGER WIN, INTEGER IERROR)
MPI_Win_create_errhandler	int MPI_Win_create_errhandler (MPI_Win_errhandler_fn *function, MPI_Errhandler *errhandler);
MPI::Win::Create_errhandler	MPI::Errhandler MPI::Win::Create_errhandler(MPI::Win::Errhandler_fn* function);
MPI_WIN_CREATE_ERRHANDLER	MPI_WIN_CREATE_ERRHANDLER(EXTERNAL FUNCTION, INTEGER ERRHANDLER, INTEGER IERROR)
MPI_Win_create_keyval	int MPI_Win_create_keyval (MPI_Win_copy_attr_function *win_copy_attr_fn, MPI_Win_delete_attr_function *win_delete_attr_fn, int *win_keyval, void *extra_state);
MPI::Win::Create_keyval	static int MPI::Win::Create_keyval(MPI::Win::Copy_attr_function* win_copy_attr_fn, MPI::Win::Delete_attr_function* win_delete_attr_fn, void* extra_state);
MPI_WIN_CREATE_KEYVAL	MPI_WIN_CREATE_KEYVAL(EXTERNAL WIN_COPY_ATTR_FN, EXTERNAL WIN_DELETE_ATTR_FN, INTEGER WIN_KEYVAL, INTEGER EXTRA_STATE, INTEGER IERROR)
MPI_Win_delete_attr	int MPI_Win_delete_attr (MPI_Win win, int win_keyval);
MPI::Win::Delete_attr	void MPI::Win::Delete_attr(int win_keyval);
MPI_WIN_DELETE_ATTR	MPI_WIN_DELETE_ATTR(INTEGER WIN, INTEGER WIN_KEYVAL, INTEGER IERROR)
MPI_Win_fence	int MPI_Win_fence (int assert, MPI_Win win);
MPI::Win::Fence	void MPI::Win::Fence(int assert) const;
MPI_WIN_FENCE	MPI_WIN_FENCE(INTEGER ASSERT, INTEGER WIN, INTEGER IERROR)
MPI_Win_free	int MPI_Win_free (MPI_Win *win);
MPI::Win::Free	void MPI::Win::Free();
MPI_WIN_FREE	MPI_WIN_FREE(INTEGER WIN, INTEGER IERROR)
MPI_Win_free_keyval	int MPI_Win_free_keyval (int *win_keyval);
MPI::Win::Free_keyval	void MPI::Win::Free_keyval(int& win_keyval);

Table 37. Bindings for one-sided communication (continued)

Subroutine name: C C++ FORTRAN	Binding: C C++ FORTRAN
MPI_WIN_FREE_KEYVAL	MPI_WIN_FREE_KEYVAL(INTEGER WIN_KEYVAL, INTEGER IERROR)
MPI_Win_get_attr	int MPI_Win_get_attr (MPI_Win win, int win_keyval, void *attribute_val, int *flag);
MPI::Win::Get_attr	bool MPI::Win::Get_attr(int win_keyval, void* attribute_val) const;
MPI_WIN_GET_ATTR	MPI_WIN_GET_ATTR(INTEGER WIN, INTEGER WIN_KEYVAL, INTEGER ATTRIBUTE_VAL, LOGICAL FLAG, INTEGER IERROR)
MPI_Win_get_errhandler	int MPI_Win_get_errhandler (MPI_Win win, MPI_Errhandler *errhandler);
MPI::Win::Get_errhandler	MPI::Errhandler MPI::Win::Get_errhandler() const;
MPI_WIN_GET_ERRHANDLER	MPI_WIN_GET_ERRHANDLER(INTEGER WIN, INTEGER ERRHANDLER, INTEGER IERROR)
MPI_Win_get_group	int MPI_Win_get_group (MPI_Win *win, MPI_Group *group);
MPI::Win::Get_group	MPI::Group MPI::Win::Get_group() const;
MPI_WIN_GET_GROUP	MPI_WIN_GET_GROUP(INTEGER WIN, INTEGER GROUP, INTEGER IERROR)
MPI_Win_lock	int MPI_Win_lock (int lock_type, int rank, int assert, MPI_Win win);
MPI::Win::Lock	void MPI::Win::Lock(int lock_type, int rank, int assert) const;
MPI_WIN_LOCK	MPI_WIN_LOCK(INTEGER LOCK_TYPE, INTEGER RANK, INTEGER ASSERT, INTEGER WIN, INTEGER IERROR)
MPI_Win_post	int MPI_Win_post (MPI_Group group, int assert, MPI_Win win);
MPI::Win::Post	void MPI::Win::Post(const MPI::Group& group, int assert) const;
MPI_WIN_POST	MPI_WIN_POST(INTEGER GROUP, INTEGER ASSERT, INTEGER WIN, INTEGER IERROR)
MPI_Win_set_attr	int MPI_Win_set_attr (MPI_Win win, int win_keyval, void *attribute_val);
MPI::Win::Set_attr	void MPI::Win::Set_attr(int win_keyval, const void* attribute_val);
MPI_WIN_SET_ATTR	MPI_WIN_SET_ATTR(INTEGER WIN, INTEGER WIN_KEYVAL, INTEGER ATTRIBUTE_VAL, INTEGER IERROR)
MPI_Win_set_errhandler	int MPI_Win_set_errhandler (MPI_Win win, MPI_Errhandler errhandler);
MPI::Win::Set_errhandler	void MPI::Win::Set_errhandler(const MPI::Errhandler& errhandler);
MPI_WIN_SET_ERRHANDLER	MPI_WIN_SET_ERRHANDLER(INTEGER WIN, INTEGER ERRHANDLER, INTEGER IERROR)
MPI_Win_start	int MPI_Win_start (MPI_Group group, int assert, MPI_Win win);
MPI::Win::Start	void MPI::Win::Start(const MPI::Group& group, int assert) const;
MPI_WIN_START	MPI_WIN_START(INTEGER GROUP, INTEGER ASSERT, INTEGER WIN, INTEGER IERROR)
MPI_Win_test	int MPI_Win_test (MPI_Win win, int *flag);
MPI::Win::Test()	bool MPI::Win::Test() const;

Table 37. Bindings for one-sided communication (continued)

Subroutine name: C C++ FORTRAN	Binding: C C++ FORTRAN
MPI_WIN_TEST	MPI_WIN_TEST(INTEGER WIN, LOGICAL FLAG, INTEGER IERROR)
MPI_Win_unlock	int MPI_Win_unlock (int rank, MPI_Win win);
MPI::Win::Unlock	void MPI::Win::Unlock(int rank) const;
MPI_WIN_UNLOCK	MPI_WIN_UNLOCK(INTEGER RANK, INTEGER WIN, INTEGER IERROR)
MPI_Win_wait	int MPI_Win_wait (MPI_Win win);
MPI::Win::Wait	void MPI::Win::Wait() const;
MPI_WIN_WAIT	MPI_WIN_WAIT(INTEGER WIN, INTEGER IERROR)

## Bindings for point-to-point communication

Table 38 lists the bindings for point-to-point communication subroutines.

Table 38. Bindings for point-to-point communication

Subroutine name: C C++ FORTRAN	Binding: C C++ FORTRAN
MPI_Bsend	int MPI_Bsend(void* buf,int count,MPI_Datatype datatype,int dest,int tag,MPI_Comm comm);
MPI::Comm::Bsend	void MPI::Comm::Bsend(const void* buf, int count, const MPI::Datatype& datatype, int dest, int tag) const;
MPI_BSEND	MPI_BSEND(CHOICE BUF,INTEGER COUNT,INTEGER DATATYPE,INTEGER DEST, INTEGER TAG,INTEGER COMM,INTEGER IERROR)
MPI_Bsend_init	int MPI_Bsend_init(void* buf,int count,MPI_Datatype datatype,int dest,int tag,MPI_Comm comm,MPI_Request *request);
MPI::Comm::Bsend_init	MPI::Prequest MPI::Comm::Bsend_init(const void* buf, int count, const MPI::Datatype& datatype, int dest, int tag) const;
MPI_BSEND_INIT	MPI_SEND_INIT(CHOICE BUF,INTEGER COUNT,INTEGER DATATYPE,INTEGER DEST,INTEGER TAG,INTEGER COMM,INTEGER REQUEST,INTEGER IERROR)
MPI_Buffer_attach	int MPI_Buffer_attach(void* buffer,int size);
MPI::Attach_buffer	void MPI::Attach_buffer(void* buffer, int size);
MPI_BUFFER_ATTACH	MPI_BUFFER_ATTACH(CHOICE BUFFER,INTEGER SIZE,INTEGER IERROR)
MPI_Buffer_detach	int MPI_Buffer_detach(void* buffer,int* size);
MPI::Detach_buffer	int MPI::Detach_buffer(void*& buffer);
MPI_BUFFER_DETACH	MPI_BUFFER_DETACH(CHOICE BUFFER,INTEGER SIZE,INTEGER IERROR)
MPI_Cancel	int MPI_Cancel(MPI_Request *request);
MPI::Request::Cancel	void MPI::Request::Cancel(void) const;

Table 38. Bindings for point-to-point communication (continued)

Subroutine name: C C++ FORTRAN	Binding: C C++ FORTRAN
MPI_CANCEL	MPI_CANCEL(INTEGER REQUEST,INTEGER IERROR)
MPI_Get_count	int MPI_Get_count(MPI_Status *status, MPI_Datatype datatype, int *count);
MPI::Status::Get_count	int MPI::Status::Get_count(const MPI::Datatype& datatype) const;
MPI_GET_COUNT	MPI_GET_COUNT(INTEGER STATUS(MPI_STATUS_SIZE),,INTEGER DATATYPE,INTEGER COUNT, INTEGER IERROR)
MPI_Ibsend	int MPI_Ibsend(void* buf,int count,MPI_Datatype datatype,int dest,int tag,MPI_Comm comm,MPI_Request *request);
MPI::Comm::Ibsend	MPI::Request MPI::Comm::Ibsend(const void* buf, int count, const MPI::Datatype& datatype, int dest, int tag) const;
MPI_IBSEND	MPI_IBSEND(CHOICE BUF,INTEGER COUNT,INTEGER DATATYPE,INTEGER DEST,INTEGER TAG,INTEGER COMM,INTEGER REQUEST,INTEGER IERROR)
MPI_Iprobe	int MPI_Iprobe(int source,int tag,MPI_Comm comm,int *flag,MPI_Status *status);
MPI::Comm::Iprobe	bool MPI::Comm::Iprobe(int source, int tag) const;
MPI_IPROBE	MPI_IPROBE(INTEGER SOURCE,INTEGER TAG,INTEGER COMM,INTEGER FLAG,INTEGER STATUS(MPI_STATUS_SIZE),INTEGER IERROR)
MPI_Irecv	int MPI_Irecv(void* buf,int count,MPI_Datatype datatype,int source,int tag,MPI_Comm comm,MPI_Request *request);
MPI::Comm::Irecv	MPI::Request MPI::Comm::Irecv(void *buf, int count, const MPI::Datatype& datatype, int source, int tag) const;
MPI_IRECV	MPI_IRECV(CHOICE BUF,INTEGER COUNT,INTEGER DATATYPE,INTEGER SOURCE,INTEGER TAG,INTEGER COMM,INTEGER REQUEST,INTEGER IERROR)
MPI_Irsend	int MPI_Irsend(void* buf,int count,MPI_Datatype datatype,int dest,int tag,MPI_Comm comm,MPI_Request *request);
MPI::Comm::Irsend	MPI::Request MPI::Comm::Irsend(const void *buf, int count, const MPI::Datatype& datatype, int dest, int tag) const;
MPI_IRSEND	MPI_IRSEND(CHOICE BUF,INTEGER COUNT,INTEGER DATATYPE,INTEGER DEST,INTEGER TAG,INTEGER COMM,INTEGER REQUEST,INTEGER IERROR)
MPI_Isend	int MPI_Isend(void* buf,int count,MPI_Datatype datatype,int dest,int tag,MPI_Comm comm,MPI_Request *request);
MPI::Comm::Isend	MPI::Request MPI::Comm::Isend(const void *buf, int count, const MPI::Datatype& datatype, int dest, int tag) const;
MPI_ISEND	MPI_ISEND(CHOICE BUF,INTEGER COUNT,INTEGER DATATYPE,INTEGER DEST,INTEGER TAG,INTEGER COMM,INTEGER REQUEST,INTEGER IERROR)
MPI_Issend	int MPI_Issend(void* buf,int count,MPI_Datatype datatype,int dest,int tag,MPI_Comm comm,MPI_Request *request);
MPI::Comm::Issend	MPI::Request MPI::Comm::Issend(const void *buf, int count, const MPI::Datatype& datatype, int dest, int tag) const;

Table 38. Bindings for point-to-point communication (continued)

Subroutine name: C C++ FORTRAN	Binding: C C++ FORTRAN
MPI_ISSEND	MPI_ISSEND(CHOICE BUF,INTEGER COUNT,INTEGER DATATYPE,INTEGER DEST,INTEGER TAG,INTEGER COMM,INTEGER REQUEST,INTEGER IERROR)
MPI_Probe	int MPI_Probe(int source,int tag,MPI_Comm comm,MPI_Status *status);
MPI::Comm::Probe	void MPI::Comm::Probe(int source, int tag) const;  void MPI::Comm::Probe(int source, int tag, MPI::Status& status) const;
MPI_PROBE	MPI_PROBE(INTEGER SOURCE,INTEGER TAG,INTEGER COMM,INTEGER STATUS(MPI_STATUS_SIZE), INTEGER IERROR)
MPI_Recv	int MPI_Recv(void* buf,int count,MPI_Datatype datatype,int source,int tag, MPI_Comm comm, MPI_Status *status);
MPI::Comm::Recv	void MPI::Comm::Recv(void* buf, int count, const MPI::Datatype& datatype, int source, int tag) const;  void MPI::Comm::Recv(void* buf, int count, const MPI::Datatype& datatype, int source, int tag, MPI::Status& status) const;
MPI_RECV	MPI_RECV(CHOICE BUF,INTEGER COUNT,INTEGER DATATYPE,INTEGER SOURCE, INTEGER TAG,INTEGER COMM,INTEGER STATUS(MPI_STATUS_SIZE),INTEGER IERROR)
MPI_Recv_init	int MPI_Recv_init(void* buf,int count,MPI_Datatype datatype,int source,int tag,MPI_Comm comm,MPI_Request *request);
MPI::Comm::Recv_init	MPI::Prequest MPI::Comm::Recv_init(void* buf, int count, const MPI::Datatype& datatype, int source, int tag) const;
MPI_RECV_INIT	MPI_RECV_INIT(CHOICE BUF,INTEGER COUNT,INTEGER DATATYPE,INTEGER SOURCE,INTEGER TAG,INTEGER COMM,INTEGER REQUEST,INTEGER IERROR)
MPI_Request_free	int MPI_Request_free(MPI_Request *request);
MPI::Request::Free	void MPI::Request::Free();
MPI_REQUEST_FREE	MPI_REQUEST_FREE(INTEGER REQUEST,INTEGER IERROR)
MPI_Rsend	int MPI_Rsend(void* buf,int count,MPI_Datatype datatype,int dest,int tag,MPI_Comm comm);
MPI::Comm::Rsend	void MPI::Comm::Rsend(const void* buf, int count, const MPI::Datatype& datatype, int dest, int tag) const;
MPI_RSEND	MPI_RSEND(CHOICE BUF,INTEGER COUNT,INTEGER DATATYPE,INTEGER DEST,INTEGER TAG,INTEGER COMM,INTEGER IERROR)
MPI_Rsend_init	int MPI_Rsend_init(void* buf,int count,MPI_Datatype datatype,int dest,int tag,MPI_Comm comm,MPI_Request *request);
MPI::Comm::Rsend_init	MPI::Prequest MPI::Comm::Rsend_init(const void* buf, int count, const MPI::Datatype& datatype, int dest, int tag) const;
MPI_RSEND_INIT	MPI_RSEND_INIT(CHOICE BUF,INTEGER COUNT,INTEGER DATATYPE,INTEGER DEST,INTEGER TAG,INTEGER COMM,INTEGER REQUEST,INTEGER IERROR)
MPI_Send	int MPI_Send(void* buf,int count,MPI_Datatype datatype,int dest,int tag,MPI_Comm comm);

Table 38. Bindings for point-to-point communication (continued)

Subroutine name: C C++ FORTRAN	Binding: C C++ FORTRAN
MPI::Comm::Send	void MPI::Comm::Send(const void* buf, int count, const MPI::Datatype& datatype, int dest, int tag) const;
MPI_SEND	MPI_SEND(CHOICE BUF,INTEGER COUNT,INTEGER DATATYPE,INTEGER DEST,INTEGER TAG,INTEGER COMM,INTEGER IERROR)
MPI_Send_init	int MPI_Send_init(void* buf,int count,MPI_Datatype datatype,int dest,int tag,MPI_Comm comm,MPI_Request *request);
MPI::Comm::Send_init	MPI::Prequest MPI::Comm::Send_init(const void* buf, int count, const MPI::Datatype& datatype, int dest, int tag) const;
MPI_SEND_INIT	MPI_SEND_INIT(CHOICE BUF,INTEGER COUNT,INTEGER DATATYPE,INTEGER DEST,INTEGER TAG,INTEGER COMM,INTEGER REQUEST,INTEGER IERROR)
MPI_Sendrecv	int MPI_Sendrecv(void *sendbuf,int sendcount,MPI_Datatype sendtype,int dest,int sendtag,void *recvbuf,int recvcount, MPI_Datatype recvtype,int source,int recvtag,MPI_Comm comm,MPI_Status *status);
MPI::Comm::Sendrecv	void MPI::Comm::Sendrecv(const void* sendbuf, int sendcount, const MPI::Datatype& sendtype, int dest, int sendtag, void* recvbuf, int recvcount, const MPI::Datatype& recvtype, int source, int recvtag) const;  void MPI::Comm::Sendrecv(const void* sendbuf, int sendcount, const MPI::Datatype& sendtype, int dest, int sendtag, void* recvbuf, int recvcount, const MPI::Datatype& recvtype, int source, int recvtag, MPI::Status& status) const;
MPI_SENDRECV	MPI_SENDRECV(CHOICE SENDBUF,INTEGER SENDCOUNT,INTEGER SENDTYPE,INTEGER DEST,INTEGER SENDTAG,CHOICE RECVBUFF,INTEGER RECVCOUNT,INTEGER RECVTYP,INTEGER SOURCE,INTEGER RECVTAG,INTEGER COMM,INTEGER STATUS(MPI_STATUS_SIZE),INTEGER IERROR)
MPI_Sendrecv_replace	int MPI_Sendrecv_replace(void* buf,int count,MPI_Datatype datatype,int dest,int sendtag,int source,int recvtag,MPI_Comm comm,MPI_Status *status);
MPI::Comm::Sendrecv_replace	void MPI::Comm::Sendrecv_replace(void* buf, int count, const MPI::Datatype& datatype, int dest, int sendtag, int source, int recvtag) const;  void MPI::Comm::Sendrecv_replace(void *buf, int count, const MPI::Datatype& datatype, int dest, int sendtag, int source, int recvtag, MPI::Status& status) const;
MPI_SENDRECV_REPLACE	MPI_SENDRECV_REPLACE(CHOICE BUF,INTEGER COUNT,INTEGER DATATYPE,INTEGER DEST,INTEGER SENDTAG,INTEGER SOURCE,INTEGER RECVTAG,INTEGER COMM,INTEGER STATUS(MPI_STATUS_SIZE),INTEGER IERROR)
MPI_Ssend	int MPI_Ssend(void* buf,int count,MPI_Datatype datatype,int dest,int tag,MPI_Comm comm);
MPI::Comm::Ssend	void MPI::Comm::Ssend(const void* buf, int count, const MPI::Datatype& datatype, int dest, int tag) const;
MPI_SSEND	MPI_SSEND(CHOICE BUF,INTEGER COUNT,INTEGER DATATYPE,INTEGER DEST,INTEGER TAG,INTEGER COMM,INTEGER IERROR)

Table 38. Bindings for point-to-point communication (continued)

<b>Subroutine name:</b> <b>C</b> <b>C++</b> <b>FORTRAN</b>	<b>Binding:</b> <b>C</b> <b>C++</b> <b>FORTRAN</b>
MPI_Ssend_init	int MPI_Ssend_init(void* buf,int count,MPI_Datatype datatype,int dest,int tag,MPI_Comm comm,MPI_Request *request);
MPI::Comm::Ssend_init	MPI::Prequest MPI::Comm::Ssend_init(const void* buf, int count, const MPI::Datatype& datatype, int dest, int tag) const;
MPI_SSEND_INIT	MPI_SSEND_INIT(CHOICE BUF,INTEGER COUNT,INTEGER DATATYPE,INTEGER DEST,INTEGER TAG,INTEGER COMM,INTEGER REQUEST,IERROR)
MPI_Start	int MPI_Start(MPI_Request *request);
MPI::Prequest::Start	void MPI::Prequest::Start();
MPI_START	MPI_START(INTEGER REQUEST,INTEGER IERROR)
MPI_Startall	int MPI_Startall(int count,MPI_Request *array_of_requests);
MPI::Prequest::Startall	void MPI::Prequest::Startall(int count, MPI::Prequest array_of_requests[]);
MPI_STARTALL	MPI_STARTALL(INTEGER COUNT,INTEGER ARRAY_OF_REQUESTS(*),INTEGER IERROR)
MPI_Test	int MPI_Test(MPI_Request *request,int *flag,MPI_Status *status);
MPI::Request::Test	bool MPI::Request::Test();
MPI_TEST	MPI_TEST(INTEGER REQUEST,INTEGER FLAG,INTEGER STATUS(MPI_STATUS_SIZE), INTEGER IERROR)
MPI_Test_cancelled	int MPI_Test_cancelled(MPI_Status *status,int *flag);
MPI::Status::Is_cancelled	bool MPI::Status::Is_cancelled() const;
MPI_TEST_CANCELLED	MPI_TEST_CANCELLED(INTEGER STATUS(MPI_STATUS_SIZE),INTEGER FLAG,INTEGER IERROR)
MPI_Testall	int MPI_Testall(int count,MPI_Request *array_of_requests,int *flag,MPI_Status *array_of_statuses);
MPI::Request::Testall	bool MPI::Request::Testall(int count, MPI::Request req_array[]);  bool MPI::Request::Testall(int count, MPI::Request req_array[], MPI::Status stat_array[]);
MPI_TESTALL	MPI_TESTALL(INTEGER COUNT,INTEGER ARRAY_OF_REQUESTS(*),INTEGER FLAG, INTEGER ARRAY_OF_STATUSES(MPI_STATUS_SIZE,*),INTEGER IERROR)
MPI_Testany	int MPI_Testany(int count, MPI_Request *array_of_requests, int *index, int *flag,MPI_Status *status);
MPI::Request::Testany	bool MPI::Request::Testany(int count, MPI::Request array[], int& index);  bool MPI::Request::Testany(int count, MPI::Request array[], int& index, MPI::Status& status);
MPI_TESTANY	MPI_TESTANY(INTEGER COUNT,INTEGER ARRAY_OF_REQUESTS(*),INTEGER INDEX,INTEGER FLAG,INTEGER STATUS(MPI_STATUS_SIZE), INTEGER IERROR)
MPI_Testsome	int MPI_Testsome(int incount,MPI_Request *array_of_requests,int *outcount,int *array_of_indices,MPI_Status *array_of_statuses);

Table 38. Bindings for point-to-point communication (continued)

Subroutine name: C C++ FORTRAN	Binding: C C++ FORTRAN
MPI::Request::Testsome	int MPI::Request::Testsome(int incount, MPI::Request req_array[], int array_of_indices[]);  int MPI::Request::Testsome(int incount, MPI::Request req_array[], int array_of_indices[], MPI::Status stat_array[]);
MPI_TESTSOME	MPI_TESTSOME(INTEGER INCOUNT,INTEGER ARRAY_OF_REQUESTS(*),INTEGER OUTCOUNT,INTEGER ARRAY_OF_INDICES(*),INTEGER ARRAY_OF_STATUSES(MPI_STATUS_SIZE,*),INTEGER IERROR)
MPI_Wait	int MPI_Wait(MPI_Request *request,MPI_Status *status);
MPI::Request::Wait	void MPI::Request::Wait();  void MPI::Request::Wait(MPI::Status& status);
MPI_WAIT	MPI_WAIT(INTEGER REQUEST,INTEGER STATUS(MPI_STATUS_SIZE),INTEGER IERROR)
MPI_Waitall	int MPI_Waitall(int count,MPI_Request *array_of_requests,MPI_Status *array_of_statuses);
MPI::Request::Waitall	void MPI::Request::Waitall(int count, MPI::Request req_array[]);  void MPI::Request::Waitall(int count, MPI::Request req_array[], MPI::Status stat_array[]);
MPI_WAITALL	MPI_WAITALL(INTEGER COUNT,INTEGER ARRAY_OF_REQUESTS(*),INTEGER ARRAY_OF_STATUSES(MPI_STATUS_SIZE,*), INTEGER IERROR)
MPI_Waitany	int MPI_Waitany(int count,MPI_Request *array_of_requests,int *index,MPI_Status *status);
MPI::Request::Waitany	int MPI::Request::Waitany(int count, MPI::Request array[]);  int MPI::Request::Waitany(int count, MPI::Request array[], MPI::Status& status);
MPI_WAITANY	MPI_WAITANY(INTEGER COUNT,INTEGER ARRAY_OF_REQUESTS(*),INTEGER INDEX, INTEGER STATUS(MPI_STATUS_SIZE),INTEGER IERROR)
MPI_Waitsome	int MPI_Waitsome(int incount,MPI_Request *array_of_requests,int *outcount,int *array_of_indices,MPI_Status *array_of_statuses);
MPI::Request::Waitsome	int MPI::Request::Waitsome(int incount, MPI::Request req_array[], int array_of_indices[]);  int MPI::Request::Waitsome(int incount, MPI::Request req_array[], int array_of_indices[], MPI::Status stat_array[]);
MPI_WAITSOME	MPI_WAITSOME(INTEGER INCOUNT,INTEGER ARRAY_OF_REQUESTS,INTEGER OUTCOUNT,INTEGER ARRAY_OF_INDICES(*),INTEGER ARRAY_OF_STATUSES(MPI_STATUS_SIZE,*),INTEGER IERROR)

## Binding for profiling control

Table 39 on page 214 lists the binding for profiling control.

Table 39. Binding for profiling control

<b>Subroutine name:</b> C C++ FORTRAN	<b>Binding:</b> C C++ FORTRAN
MPI_Pcontrol	int MPI_Pcontrol(const int level,...);
MPI::Pcontrol	void MPI::Pcontrol(const int level, ...);
MPI_PCONTROL	MPI_PCONTROL(INTEGER LEVEL,...)

## Bindings for topologies

Table 40 lists the bindings for topology subroutines.

Table 40. Bindings for topologies

<b>Subroutine name:</b> C C++ FORTRAN	<b>Binding:</b> C C++ FORTRAN
MPI_Cart_coords	int MPI_Cart_coords(MPI_Comm comm,int rank,int maxdims,int *coords);
MPI::Cartcomm::Get_coords	void MPI::Cartcomm::Get_coords(int rank, int maxdims, int coords[]) const;
MPI_CART_COORDS	MPI_CART_COORDS(INTEGER COMM,INTEGER RANK,INTEGER MAXDIMS,INTEGER COORDS(*),INTEGER IERROR)
MPI_Cart_create	int MPI_Cart_create(MPI_Comm comm_old,int ndims,int *dims,int *periods,int reorder,MPI_Comm *comm_cart);
MPI::Intracomm::Create_cart	MPI::Cartcomm MPI::Intracomm::Create_cart(int ndims, const int dims[], const bool periods[], bool reorder) const;
MPI_CART_CREATE	MPI_CART_CREATE(INTEGER COMM_OLD,INTEGER NDIMS,INTEGER DIMS(*), INTEGER PERIODS(*),INTEGER REORDER,INTEGER COMM_CART,INTEGER IERROR)
MPI_Cart_get	int MPI_Cart_get(MPI_Comm comm,int maxdims,int *dims,int *periods,int *coords);
MPI::Cartcomm::Get_topo	void MPI::Cartcomm::Get_topo(int maxdims, int dims[], bool periods[], int coords[]) const;
MPI_CART_GET	MPI_CART_GET(INTEGER COMM,INTEGER MAXDIMS,INTEGER DIMS(*),INTEGER PERIODS(*),INTEGER COORDS(*),INTEGER IERROR)
MPI_Cart_map	int MPI_Cart_map(MPI_Comm comm,int ndims,int *dims,int *periods,int *newrank);
MPI::Cartcomm::Map	int MPI::Cartcomm::Map(int ndims, const int dims[], const bool periods[]) const;
MPI_CART_MAP	MPI_CART_MAP(INTEGER COMM,INTEGER NDIMS,INTEGER DIMS(*),INTEGER PERIODS(*),INTEGER NEWRANK,INTEGER IERROR)
MPI_Cart_rank	int MPI_Cart_rank(MPI_Comm comm,int *coords,int *rank);
MPI::Cartcomm::Get_cart_rank	int MPI::Cartcomm::Get_cart_rank(const int coords[]) const;
MPI_CART_RANK	MPI_CART_RANK(INTEGER COMM,INTEGER COORDS(*),INTEGER RANK,INTEGER IERROR)

Table 40. Bindings for topologies (continued)

Subroutine name: C C++ FORTRAN	Binding: C C++ FORTRAN
MPI_Cart_shift	int MPI_Cart_shift(MPI_Comm comm,int direction,int disp,int *rank_source,int *rank_dest);
MPI::Cartcomm::Shift	void MPI::Cartcomm::Shift(int direction, int disp, int &rank_source, int &rank_dest) const;
MPI_CART_SHIFT	MPI_CART_SHIFT(INTEGER COMM,INTEGER DIRECTION,INTEGER DISP, INTEGER RANK_SOURCE,INTEGER RANK_DEST,INTEGER IERROR)
MPI_Cart_sub	int MPI_Cart_sub(MPI_Comm comm,int *remain_dims,MPI_Comm *newcomm);
MPI::Cartcomm::Sub	MPI::Cartcomm MPI::Cartcomm::Sub(const bool remain_dims[]) const;
MPI_CART_SUB	MPI_CART_SUB(INTEGER COMM,INTEGER REMAIN_DIMS,INTEGER NEWCOMM, INTEGER IERROR)
MPI_Cartdim_get	int MPI_Cartdim_get(MPI_Comm comm, int *ndims);
MPI::Cartcomm::Get_dim	int MPI::Cartcomm::Get_dim() const;
MPI_CARTDIM_GET	MPI_CARTDIM_GET(INTEGER COMM,INTEGER NDIMS,INTEGER IERROR)
MPI_Dims_create	int MPI_Dims_create(int nnodes,int ndims,int *dims);
MPI::Compute_dims	void MPI::Compute_dims(int nnodes, int ndims, int dims[]);
MPI_DIMS_CREATE	MPI_DIMS_CREATE(INTEGER NNODES,INTEGER NDIMS,INTEGER DIMS(*), INTEGER IERROR)
MPI_Graph_create	int MPI_Graph_create(MPI_Comm comm_old,int nnodes,int *index,int *edges,int reorder,MPI_Comm *comm_graph);
MPI::Intracomm::Create_graph	MPI::Graphcomm MPI::Intracomm::Create_graph(int nnodes, const int index[], const int edges[], bool reorder) const;
MPI_GRAPH_CREATE	MPI_GRAPH_CREATE(INTEGER COMM_OLD,INTEGER NNODES,INTEGER INDEX(*), INTEGER EDGES(*),INTEGER REORDER,INTEGER COMM_GRAPH,INTEGER IERROR)
MPI_Graph_get	int MPI_Graph_get(MPI_Comm comm,int maxindex,int maxedges,int *index, int *edges);
MPI::Graphcomm::Get_topo	void MPI::Graphcomm::Get_topo(int maxindex, int maxedges, int index[], int edges[]) const;
MPI_GRAPH_GET	MPI_GRAPH_GET(INTEGER COMM,INTEGER MAXINDEX,INTEGER MAXEDGES,INTEGER INDEX(*),INTEGER EDGES(*),INTEGER IERROR)
MPI_Graph_map	int MPI_Graph_map(MPI_Comm comm,int nnodes,int *index,int *edges,int *newrank);
MPI::Graphcomm::Map	int MPI::Graphcomm::Map(int nnodes, const int index[], const int edges[]) const;
MPI_GRAPH_MAP	MPI_GRAPH_MAP(INTEGER COMM,INTEGER NNODES,INTEGER INDEX(*),INTEGER EDGES(*),INTEGER NEWRANK,INTEGER IERROR)
MPI_Graph_neighbors	int MPI_Graph_neighbors(MPI_Comm comm,int rank,int maxneighbors,int *neighbors);
MPI::Graphcomm::Get_neighbors	void MPI::Graphcomm::Get_neighbors(int rank, int maxneighbors, int neighbors[]) const;

Table 40. Bindings for topologies (continued)

Subroutine name: C C++ FORTRAN	Binding: C C++ FORTRAN
MPI_GRAPH_NEIGHBORS	MPI_GRAPH_NEIGHBORS(MPI_COMM COMM,INTEGER RANK,INTEGER MAXNEIGHBORS,INTEGER NNEIGHBORS(*),INTEGER IERROR)
MPI_Graph_neighbors_count	int MPI_Graph_neighbors_count(MPI_Comm comm,int rank,int *nneighbors);
MPI::Graphcomm::Get_neighbors_count	int MPI::Graphcomm::Get_neighbors_count(int rank) const;
MPI_GRAPH_NEIGHBORS_COUNT	MPI_GRAPH_NEIGHBORS_COUNT(INTEGER COMM,INTEGER RANK,INTEGER NEIGHBORS, INTEGER IERROR)
MPI_Graphdims_get	int MPI_Graphdims_get(MPI_Comm comm,int *nnodes,int *nedges);
MPI::Graphcomm::Get_dims	void MPI::Graphcomm::Get_dims(int nnodes[], int nedges[]) const;
MPI_GRAPHDIMS_GET	MPI_GRAPHDIMS_GET(INTEGER COMM,INTEGER NNODES,INTEGER NEDGES, INTEGER IERROR)
MPI_Topo_test	int MPI_Topo_test(MPI_Comm comm,int *status);
MPI::Comm::Get_topology	int MPI::Comm::Get_topology() const;
MPI_TOPO_TEST	MPI_TOPO_TEST(INTEGER COMM,INTEGER STATUS,INTEGER IERROR)

---

## Appendix E. PE MPI buffer management for eager protocol

The Parallel Environment implementation of MPI uses an **eager send** protocol for messages whose size is up to the **eager limit**. This value can be allowed to default, or can be specified with the **MP\_EAGER\_LIMIT** environment variable or the **-eager\_limit** command-line flag. In an eager send, the entire message is sent immediately to its destination and the send buffer is returned to the application. Since the message is sent without knowing if there is a matching receive waiting, the message may need to be stored in the early arrival buffer at the destination, until a matching receive is posted by the application. The MPI standard requires that an eager send be done only if it can be guaranteed that there is sufficient buffer space. If a send is posted at some source (sender) when buffer space cannot be guaranteed, the send must not complete at the source until it is known that there will be a place for the message at the destination.

PE MPI uses a **credit flow control**, by which senders track the buffer space that can be guaranteed at each destination. For each source-destination pair, an eager send consumes a **message credit** at the source, and a match at the destination generates a message credit. The message credits generated at the destination are returned to the sender to enable additional eager sends. The message credits are returned piggyback on an application message when possible. If there is no return traffic, they will accumulate at the destination until their number reaches some threshold, and then be sent back as a batch to minimize network traffic. When a sender has no message credits, its sends must proceed using **rendezvous protocol** until message credits become available. The fallback to rendezvous protocol may impact performance. With a reasonable supply of message credits, most applications will find that the credits return soon enough to enable messages that are not larger than the eager limit to continue to be sent eagerly.

Assuming a pre-allocated early arrival buffer (whose size cannot increase), the number of message credits that the early arrival buffer represents is equal to the early arrival buffer size divided by the eager limit. Since no sender can know how many other tasks will also send eagerly to a given destination, the message credits must be divided among sender tasks equally. If every task sends eagerly to a single destination that is not posting receives, each sender consumes its message credits, fills its share of the destination early arrival buffer, and reverts to rendezvous protocol. This prevents an overflow at the destination, which would result in job failure. To offer a reasonable number of message credits per source-destination pair at larger task counts, either a very large pre-allocated early arrival buffer, or a very small eager limit is needed.

It would be unusual for a real application to flood a single destination this way, and well-written applications try to pre-post their receives. An eager send must consume a message credit at the send side, but when the message arrives and matches a waiting receive, it does not consume any of the early arrival buffer space. The message credit is available to be returned to the sender, but does not return instantly. When they pre-post and do not flood, real applications seldom use more than a small percentage of the total early arrival buffer space. However, because message credits must be managed for the worst case, they may be depleted at the send side. The send side then reverts to rendezvous protocol, even though there is plenty of early arrival buffer space available, or there is a matching receive waiting at the receive side, which would then not need to use the early arrival buffer.

The advantage of a pre-allocated early arrival buffer is that the Parallel Environment implementation of MPI is able to allocate and free early arrival space in the pre-allocated buffer quickly, and because the space is owned by the MPI library, it is certain to be available if needed. There is nothing an application can do to make the space that is promised by message credits unavailable in the event that all message credits are used. A disadvantage is that the space that is pre-allocated to the early arrival buffer to support adequate message credits is denied to the application, even if only a small portion of that pre-allocated space is ever used.

With PE 4.2, MPI users are given new control over buffer pre-allocation and message credits. MPI users can specify both a pre-allocated and maximum early arrival buffer size. The pre-allocated early arrival buffer is set aside for efficient management, and guaranteed availability. If the early arrival buffer requirement exceeds the pre-allocated space, extra early arrival buffer space comes from the heap using **malloc** and **free**. Message credits are calculated based on the maximum buffer size, and all of the pre-allocated early arrival buffer is used before using **malloc** and **free**. Since message credits are based on the maximum buffer size, an application that floods a single destination with unmatched eager messages from all senders, could require the specified maximum. If other heap usage has made that space unavailable, a **malloc** could fail and the job would be terminated. However, well-designed applications might see better performance from additional credits, but may not even fill the pre-allocated early arrival buffer, let alone come near needing the promised maximum. An omitted maximum, or any value at or below the **pre\_allocated\_size**, will cause message credits to be limited so that there will never be an overflow of the pre-allocated early arrival buffer.

For most applications, the default value for the early arrival buffer should be satisfactory, and with the default, the message credits are calculated based on the pre-allocated size. The pre-allocated size can be changed from its default by setting the **MP\_BUFFER\_MEM** environment variable or using the **-buffer\_mem** command-line flag with a single value. The message credits are calculated based on the modified pre-allocated size. There will be no use of **malloc** and **free** after initialization (**MPI\_Init**). This is the way earlier versions of the Parallel Environment implementation of MPI worked, so there is no need to learn new habits for command-line arguments, or to make changes to existing run scripts and default shell environments.

For some applications, in particular those that are memory constrained or run at large task counts, it may be useful to adjust the size of the pre-allocated early arrival buffer to slightly more than the application's peak demand, but specify a higher maximum early arrival buffer size so that enough message credits are available to ensure few or no fallbacks to rendezvous protocol. For a given run, you can use the **MP\_STATISTICS** environment variable to see how much early arrival buffer space is used at peak demand, and how often a send that is small enough to be an eager send, was processed using rendezvous protocol due to a message credit shortage.

By decreasing the pre-allocated early arrival buffer size to slightly larger than the application's peak demand, you avoid wasting pre-allocated buffer space. By increasing the maximum buffer size, you provide credits which can reduce or eliminate fallbacks to rendezvous protocol. The application's peak demand and fallback frequency can vary from run to run, and the amount of variation may depend on the nature of the application. If the application's peak demand is larger than the pre-allocated early arrival buffer size, the use of **malloc** and **free** may cause a performance impact. The credit flow control will guarantee that the

| application's peak demand will never exceed the specified maximum. However, if  
| you pick a maximum that cannot be satisfied, it is possible for an MPI application  
| that does aggressive but valid flooding of a single destination to fail in a malloc.

| The risk of needing the maximum early arrival buffer size is small in  
| well-structured applications, so with very large task counts, you may choose to set  
| an unrealistic maximum to allow a higher eager limit and get enough message  
| credits to maximize performance. However, be aware that if the application  
| behaves differently than expected and requires significantly more storage than the  
| pre-allocated early arrival buffer size, and this storage is not available before  
| message credit shortages throttle eager sending, unexpected paging or even malloc  
| failures are possible. (To **throttle** a car engine is to choke off its air and fuel intake  
| by lifting your foot from the gas pedal when you want to keep the car from going  
| faster than you can control).



---

## Appendix F. Accessibility

Accessibility features help a user who has a physical disability, such as restricted mobility or limited vision, to use software products successfully. The major accessibility features enable users to:

- Use assistive technologies such as screen readers and screen magnifier software
- Operate specific or equivalent features using only the keyboard
- Customize display attributes such as color, contrast, and font size.

---

### Accessibility information

Accessibility information for IBM products is available online. Visit the IBM Accessibility Center at:

<http://www.ibm.com/able/>

To request accessibility information, click **Product accessibility information**.

---

### Using assistive technologies

Assistive technology products, such as screen readers, function with user interfaces. Consult the assistive technology documentation for specific information when using such products to access interfaces.



---

## Notices

This information was developed for products and services offered in the U.S.A.

IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing  
IBM Corporation  
North Castle Drive  
Armonk, NY 10504-1785  
U.S.A.

For license inquiries regarding double-byte (DBCS) information, contact the IBM Intellectual Property Department in your country or send inquiries, in writing, to:

IBM World Trade Asia Corporation  
Licensing  
2-31 Roppongi 3-chome, Minato-ku  
Tokyo 106, Japan

The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law:

INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Any references in this information to non-IBM Web sites are provided for convenience only and do not in any manner serve as an endorsement of those Web sites. The materials at those Web sites are not part of the materials for this IBM product and use of those Web sites is at your own risk.

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you.

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact:

IBM Corporation  
Department LJEB/P905  
2455 South Road  
Poughkeepsie, NY 12601-5400  
U.S.A

Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

The licensed program described in this document and all licensed material available for it are provided by IBM under terms of the IBM Customer Agreement, IBM International Program License Agreement or any equivalent agreement between us.

Any performance data contained herein was determined in a controlled environment. Therefore, the results obtained in other operating environments may vary significantly. Some measurements may have been made on development-level systems and there is no guarantee that these measurements will be the same on generally available systems. Furthermore, some measurement may have been estimated through extrapolation. Actual results may vary. Users of this document should verify the applicable data for their specific environment.

Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not tested those products and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

All statements regarding IBM's future direction or intent are subject to change or withdrawal without notice, and represent goals and objectives only.

This information contains examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples include the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to the names and addresses used by an actual business enterprise is entirely coincidental.

#### COPYRIGHT LICENSE:

This information contains sample application programs in source language, which illustrates programming techniques on various operating platforms. You may copy, modify, and distribute these sample programs in any form without payment to IBM, for the purposes of developing, using, marketing or distributing application programs conforming to the application programming interface for the operating platform for which the sample programs are written. These examples have not been thoroughly tested under all conditions. IBM, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs. You may copy, modify, and distribute these sample programs in any form without payment to

IBM for the purposes of developing, using, marketing, or distributing application programs conforming to IBM's application programming interfaces.

Each copy or any portion of these sample programs or any derivative work, must include a copyright notice as follows:

© (your company name) (year). Portions of this code are derived from IBM Corp. Sample Programs. © Copyright IBM Corp. \_enter the year or years\_. All rights reserved.

All implemented function in the PE MPI product is designed to comply with the requirements of the Message Passing Interface Forum, MPI: A Message-Passing Interface Standard. The standard is documented in two volumes, Version 1.1, University of Tennessee, Knoxville, Tennessee, June 6, 1995 and *MPI-2: Extensions to the Message-Passing Interface*, University of Tennessee, Knoxville, Tennessee, July 18, 1997. The second volume includes a section identified as MPI 1.2 with clarifications and limited enhancements to MPI 1.1. It also contains the extensions identified as MPI 2.0. The three sections, MPI 1.1, MPI 1.2 and MPI 2.0 taken together constitute the current standard for MPI.

PE MPI provides support for all of MPI 1.1 and MPI 1.2. PE MPI also provides support for all of the MPI 2.0 Enhancements, except the contents of the chapter titled *Process Creation and Management*.

If you believe that PE MPI does not comply with the MPI standard for the portions that are implemented, please contact IBM Service.

---

## Trademarks

The following are trademarks of International Business Machines Corporation in the United States, other countries, or both:

AFS  
AIX  
AIX 5L  
AIXwindows<sup>®</sup>  
DFS  
e (logo)  
IBM  
IBM (logo)  
IBMLink<sup>™</sup>  
LoadLeveler  
POWER<sup>™</sup>  
POWER3  
POWER4  
POWER5  
pSeries  
RS/6000  
SP

UNIX is a registered trademark of The Open Group in the United States and other countries.

| Linux is a trademark of Linus Torvalds in the United States, other countries, or  
| both.

Other company, product, and service names may be trademarks or service marks of others.

---

## Acknowledgments

The PE Benchmark product includes software developed by the Apache Software Foundation, <http://www.apache.org>.

---

# Glossary

## A

**AFS.** Andrew File System.

**address.** A value, possibly a character or group of characters that identifies a register, a device, a particular part of storage, or some other data source or destination.

**AIX.** Abbreviation for Advanced Interactive Executive, IBM's licensed version of the UNIX operating system. AIX is particularly suited to support technical computing applications, including high-function graphics and floating-point computations.

**AIXwindows Environment/6000.** A graphical user interface (GUI) for the IBM RS/6000. It has the following components:

- A graphical user interface and toolkit based on OSF/Motif
- Enhanced X-Windows, an enhanced version of the MIT X Window System
- Graphics Library (GL), a graphical interface library for the application programmer that is compatible with Silicon Graphics' GL interface.

**API.** Application programming interface.

**application.** The use to which a data processing system is put; for example, a payroll application, an airline reservation application.

**argument.** A parameter passed between a calling program and a called program or subprogram.

**attribute.** A named property of an entity.

**Authentication.** The process of validating the identity of a user or server.

**Authorization.** The process of obtaining permission to perform specific actions.

## B

**bandwidth.** The difference, expressed in hertz, between the highest and the lowest frequencies of a range of frequencies. For example, analog transmission by recognizable voice telephone requires a bandwidth of about 3000 hertz (3 kHz). The bandwidth of an optical link designates the information-carrying capacity of the link and is related to the maximum bit rate that a fiber link can support.

**blocking operation.** An operation that does not complete until the operation either succeeds or fails. For example, a blocking receive will not return until a

message is received or until the channel is closed and no further messages can be received.

**breakpoint.** A place in a program, specified by a command or a condition, where the system halts execution and gives control to the workstation user or to a specified program.

**broadcast operation.** A communication operation where one processor sends (or broadcasts) a message to all other processors.

**buffer.** A portion of storage used to hold input or output data temporarily.

## C

**C.** A general-purpose programming language. It was formalized by Uniform in 1983 and the ANSI standards committee for the C language in 1984.

**C++.** A general-purpose programming language that is based on the C language. C++ includes extensions that support an object-oriented programming paradigm. Extensions include:

- strong typing
- data abstraction and encapsulation
- polymorphism through function overloading and templates
- class inheritance.

**chaotic relaxation.** An iterative relaxation method that uses a combination of the Gauss-Seidel and Jacobi-Seidel methods. The array of discrete values is divided into subregions that can be operated on in parallel. The subregion boundaries are calculated using the Jacobi-Seidel method, while the subregion interiors are calculated using the Gauss-Seidel method. See also *Gauss-Seidel*.

**client.** A function that requests services from a server and makes them available to the user.

| **cluster.** A group of processors interconnected through  
| a high-speed network that can be used for  
| high-performance computing.

**Cluster 1600.** See IBM @server Cluster 1600.

**collective communication.** A communication operation that involves more than two processes or tasks. Broadcasts, reductions, and the **MPI\_Allreduce** subroutine are all examples of collective communication operations. All tasks in a communicator must participate.

**command alias.** When using the PE command-line debugger **pdbx**, you can create abbreviations for existing commands using the **pdbx alias** command. These abbreviations are known as *command aliases*.

**Communication Subsystem (CSS).** A component of the IBM Parallel System Support Programs for AIX that provides software support for the high performance switch. CSS provides two protocols: Internet Protocol (IP) for LAN-based communication and User Space protocol as a message passing interface that is optimized for performance over the switch. See also *Internet Protocol* and *User Space*.

**communicator.** An MPI object that describes the communication context and an associated group of processes.

**compile.** To translate a source program into an executable program.

**condition.** One of a set of specified values that a data item can assume.

**control workstation.** A workstation attached to the IBM RS/6000 SP that serves as a single point of control allowing the administrator or operator to monitor and manage the system using IBM Parallel System Support Programs for AIX.

**core dump.** A process by which the current state of a program is preserved in a file. Core dumps are usually associated with programs that have encountered an unexpected, system-detected fault, such as a Segmentation Fault or a severe user error. The current program state is needed for the programmer to diagnose and correct the problem.

**core file.** A file that preserves the state of a program, usually just before a program is terminated for an unexpected error. See also *core dump*.

**current context.** When using the **pdbx** debugger, control of the parallel program and the display of its data can be limited to a subset of the tasks belonging to that program. This subset of tasks is called the *current context*. You can set the current context to be a single task, multiple tasks, or all the tasks in the program.

## D

**data decomposition.** A method of breaking up (or decomposing) a program into smaller parts to exploit parallelism. One divides the program by dividing the data (usually arrays) into smaller parts and operating on each part independently.

**data parallelism.** Refers to situations where parallel tasks perform the same computation on different sets of data.

**dbx.** A symbolic command-line debugger that is often provided with UNIX systems. The PE command-line debugger **pdbx** is based on the **dbx** debugger.

**debugger.** A debugger provides an environment in which you can manually control the execution of a program. It also provides the ability to display the program's data and operation.

**distributed shell (dsh).** An IBM Parallel System Support Programs for AIX command that lets you issue commands to a group of hosts in parallel. See *IBM Parallel System Support Programs for AIX: Command and Technical Reference* for details.

**domain name.** The hierarchical identification of a host system (in a network), consisting of human-readable labels, separated by decimal points.

**DPCL target application.** The executable program that is instrumented by a Dynamic Probe Class Library (DPCL) analysis tool. It is the process (or processes) into which the DPCL analysis tool inserts probes. A target application could be a serial or parallel program. Furthermore, if the target application is a parallel program, it could follow either the SPMD or the MPMD model, and may be designed for either a message-passing or a shared-memory system.

## E

**environment variable.** (1) A variable that describes the operating environment of the process. Common environment variables describe the home directory, command search path, and the current time zone. (2) A variable that is included in the current software environment and is therefore available to any called program that requests it.

**Ethernet.** A baseband local area network (LAN) that allows multiple stations to access the transmission medium at will without prior coordination, avoids contention by using carrier sense and deference, and resolves contention by using collision detection and delayed retransmission. Ethernet uses carrier sense multiple access with collision detection (CSMA/CD).

**event.** An occurrence of significance to a task — the completion of an asynchronous operation such as an input/output operation, for example.

**executable.** A program that has been link-edited and therefore can be run in a processor.

**execution.** To perform the actions specified by a program or a portion of a program.

**expression.** In programming languages, a language construct for computing a value from one or more operands.

## F

**fairness.** A policy in which tasks, threads, or processes must be allowed eventual access to a resource for which they are competing. For example, if multiple threads are simultaneously seeking a lock, no set of circumstances can cause any thread to wait indefinitely for access to the lock.

**FDDI.** Fiber Distributed Data Interface.

**Fiber Distributed Data Interface (FDDI).** An American National Standards Institute (ANSI) standard for a local area network (LAN) using optical fiber cables. An FDDI LAN can be up to 100 kilometers (62 miles) long, and can include up to 500 system units. There can be up to 2 kilometers (1.24 miles) between system units and concentrators.

**file system.** In the AIX operating system, the collection of files and file management structures on a physical or logical mass storage device, such as a diskette or minidisk.

**fileset.** (1) An individually-installable option or update. Options provide specific functions. Updates correct an error in, or enhance, a previously installed program. (2) One or more separately-installable, logically-grouped units in an installation package. See also *licensed program* and *package*.

**foreign host.** See *remote host*.

**FORTRAN.** One of the oldest of the modern programming languages, and the most popular language for scientific and engineering computations. Its name is a contraction of *FORMula TRANslation*. The two most common FORTRAN versions are FORTRAN 77, originally standardized in 1978, and FORTRAN 90. FORTRAN 77 is a proper subset of FORTRAN 90.

**function cycle.** A chain of calls in which the first caller is also the last to be called. A function that calls itself recursively is not considered a function cycle.

**functional decomposition.** A method of dividing the work in a program to exploit parallelism. The program is divided into independent pieces of functionality, which are distributed to independent processors. This method is in contrast to data decomposition, which distributes the same work over different data to independent processors.

**functional parallelism.** Refers to situations where parallel tasks specialize in particular work.

## G

**Gauss-Seidel.** An iterative relaxation method for solving Laplace's equation. It calculates the general solution by finding particular solutions to a set of discrete points distributed throughout the area in

question. The values of the individual points are obtained by averaging the values of nearby points. Gauss-Seidel differs from Jacobi-Seidel in that, for the  $i+1$ st iteration, Jacobi-Seidel uses only values calculated in the  $i$ th iteration. Gauss-Seidel uses a mixture of values calculated in the  $i$ th and  $i+1$ st iterations.

**global max.** The maximum value across all processors for a given variable. It is global in the sense that it is global to the available processors.

**global variable.** A variable defined in one portion of a computer program and used in at least one other portion of the computer program.

**gprof.** A UNIX command that produces an execution profile of C, COBOL, FORTRAN, or Pascal programs. The execution profile is in a textual and tabular format. It is useful for identifying which routines use the most CPU time. See the man page on **gprof**.

**graphical user interface (GUI).** A type of computer interface consisting of a visual metaphor of a real-world scene, often of a desktop. Within that scene are icons, which represent actual objects, that the user can access and manipulate with a pointing device.

**GUI.** Graphical user interface.

## H

**high performance switch.** The high-performance message-passing network of the IBM RS/6000 SP that connects all processor nodes.

**HIPPI.** High performance parallel interface.

**hook.** A **pdbx** command that lets you re-establish control over all tasks in the current context that were previously unhooked with this command.

**home node.** The node from which an application developer compiles and runs his program. The home node can be any workstation on the LAN.

**host.** A computer connected to a network that provides an access method to that network. A host provides end-user services.

**host list file.** A file that contains a list of host names, and possibly other information, that was defined by the application that reads it.

**host name.** The name used to uniquely identify any computer on a network.

**hot spot.** A memory location or synchronization resource for which multiple processors compete excessively. This competition can cause a disproportionately large performance degradation when one processor that seeks the resource blocks, preventing many other processors from having it, thereby forcing them to become idle.

## I

**IBM @server Cluster 1600.** An IBM @server Cluster 1600 is any PSSP or CSM-managed cluster comprised of POWER microprocessor based systems (including RS/6000 SMPs, RS/6000 SP nodes, and pSeries SMPs).

**IBM Parallel Environment (PE) for AIX.** A licensed program that provides an execution and development environment for parallel C, C++, and FORTRAN programs. It also includes tools for debugging, profiling, and tuning parallel programs.

**installation image.** A file or collection of files that are required in order to install a software product on a IBM RS/6000 workstation or on SP system nodes. These files are in a form that allows them to be installed or removed with the AIX **installp** command. See also *fileset*, *licensed program*, and *package*.

**Internet.** The collection of worldwide networks and gateways that function as a single, cooperative virtual network.

**Internet Protocol (IP).** (1) The TCP/IP protocol that provides packet delivery between the hardware and user processes. (2) The SP switch library, provided with the IBM Parallel System Support Programs for AIX, that follows the IP protocol of TCP/IP.

**IP.** Internet Protocol.

## J

**Jacobi-Seidel.** See *Gauss-Seidel*.

## K

**Kerberos.** A publicly available security and authentication product that works with the IBM Parallel System Support Programs for AIX software to authenticate the execution of remote commands.

**kernel.** The core portion of the UNIX operating system that controls the resources of the CPU and allocates them to the users. The kernel is memory-resident, is said to run in *kernel mode* (in other words, at higher execution priority level than *user mode*), and is protected from user tampering by the hardware.

## L

**Laplace's equation.** A homogeneous partial differential equation used to describe heat transfer, electric fields, and many other applications.

**latency.** The time interval between the instant when an instruction control unit initiates a call for data transmission, and the instant when the actual transfer

of data (or receipt of data at the remote end) begins. Latency is related to the hardware characteristics of the system and to the different layers of software that are involved in initiating the task of packing and transmitting the data.

**licensed program.** A collection of software packages sold as a product that customers pay for to license. A licensed program can consist of packages and file sets a customer would install. These packages and file sets bear a copyright and are offered under the terms and conditions of a licensing agreement. See also *fileset* and *package*.

**lightweight corefiles.** An alternative to standard AIX corefiles. Corefiles produced in the *Standardized Lightweight Corefile Format* provide simple process stack traces (listings of function calls that led to the error) and consume fewer system resources than traditional corefiles.

**LoadLeveler.** A job management system that works with POE to let users run jobs and match processing needs with system resources, in order to make better use of the system.

**local variable.** A variable that is defined and used only in one specified portion of a computer program.

**loop unrolling.** A program transformation that makes multiple copies of the body of a loop, also placing the copies within the body of the loop. The loop trip count and index are adjusted appropriately so the new loop computes the same values as the original. This transformation makes it possible for a compiler to take additional advantage of instruction pipelining, data cache effects, and software pipelining.

See also *optimization*.

## M

| **management domain .** A set of nodes configured for  
| manageability by the Clusters Systems Management  
| (CSM) product. Such a domain has a management  
| server that is used to administer a number of managed  
| nodes. Only management servers have knowledge of  
| the whole domain. Managed nodes only know about  
| the servers managing them; they know nothing of each  
| other. Contrast with *peer domain*.

**menu.** A list of options displayed to the user by a data processing system, from which the user can select an action to be initiated.

**message catalog.** A file created using the AIX Message Facility from a message source file that contains application error and other messages, which can later be translated into other languages without having to recompile the application source code.

**message passing.** Refers to the process by which parallel tasks explicitly exchange program data.

**Message Passing Interface (MPI).** A standardized API for implementing the message-passing model.

**MIMD.** Multiple instruction stream, multiple data stream.

**Multiple instruction stream, multiple data stream (MIMD).** A parallel programming model in which different processors perform different instructions on different sets of data.

**MPMD.** Multiple program, multiple data.

**Multiple program, multiple data (MPMD).** A parallel programming model in which different, but related, programs are run on different sets of data.

**MPI.** Message Passing Interface.

## N

**network.** An interconnected group of nodes, lines, and terminals. A network provides the ability to transmit data to and receive data from other systems and users.

**Network Information Services.** A set of UNIX network services (for example, a distributed service for retrieving information about the users, groups, network addresses, and gateways in a network) that resolve naming and addressing differences among computers in a network.

**NIS.** See *Network Information Services*.

**node.** (1) In a network, the point where one or more functional units interconnect transmission lines. A computer location defined in a network. (2) In terms of the IBM RS/6000 SP, a single location or workstation in a network. An SP node is a physical entity (a processor).

**node ID.** A string of unique characters that identifies the node on a network.

**nonblocking operation.** An operation, such as sending or receiving a message, that returns immediately whether or not the operation was completed. For example, a nonblocking receive will not wait until a message is sent, but a blocking receive will wait. A nonblocking receive will return a status value that indicates whether or not a message was received.

## O

**object code.** The result of translating a computer program to a relocatable, low-level form. Object code contains machine instructions, but symbol names (such as array, scalar, and procedure names), are not yet given a location in memory. Contrast with *source code*.

**optimization.** A widely-used (though not strictly accurate) term for program performance improvement,

especially for performance improvement done by a compiler or other program translation software. An optimizing compiler is one that performs extensive code transformations in order to obtain an executable that runs faster but gives the same answer as the original. Such code transformations, however, can make code debugging and performance analysis very difficult because complex code transformations obscure the correspondence between compiled and original source code.

**option flag.** Arguments or any other additional information that a user specifies with a program name. Also referred to as *parameters* or *command-line options*.

## P

**package.** A number of file sets that have been collected into a single installable image of licensed programs. Multiple file sets can be bundled together for installing groups of software together. See also *fileset* and *licensed program*.

**parallelism.** The degree to which parts of a program may be concurrently executed.

**parallelize.** To convert a serial program for parallel execution.

**parallel operating environment (POE).** An execution environment that smooths the differences between serial and parallel execution. It lets you submit and manage parallel jobs. It is abbreviated and commonly known as POE.

**parameter.** (1) In FORTRAN, a symbol that is given a constant value for a specified application. (2) An item in a menu for which the operator specifies a value or for which the system provides a value when the menu is interpreted. (3) A name in a procedure that is used to refer to an argument that is passed to the procedure. (4) A particular piece of information that a system or application program needs to process a request.

**partition.** (1) A fixed-size division of storage. (2) In terms of the IBM RS/6000 SP, a logical collection of nodes to be viewed as one system or domain. System partitioning is a method of organizing the SP system into groups of nodes for testing or running different levels of software of product environments.

**Partition Manager.** The component of the parallel operating environment (POE) that allocates nodes, sets up the execution environment for remote tasks, and manages distribution or collection of standard input (STDIN), standard output (STDOUT), and standard error (STDERR).

**pdbx.** The parallel, symbolic command-line debugging facility of PE. **pdbx** is based on the **dbx** debugger and has a similar interface.

**PE.** The IBM Parallel Environment for AIX licensed program.

| **peer domain.** A set of nodes configured for high  
| availability by the RSCT configuration manager. Such a  
| domain has no distinguished or master node. All nodes  
| are aware of all other nodes, and administrative  
| commands can be issued from any node in the domain.  
| All nodes also have a consistent view of the domain  
| membership. Contrast with *management domain*.

**performance monitor.** A utility that displays how effectively a system is being used by programs.

**PID.** Process identifier.

**POE.** parallel operating environment.

**pool.** Groups of nodes on an SP system that are known to LoadLeveler, and are identified by a pool name or number.

**point-to-point communication.** A communication operation that involves exactly two processes or tasks. One process initiates the communication through a *send* operation. The partner process issues a *receive* operation to accept the data being sent.

**procedure.** (1) In a programming language, a block, with or without formal parameters, whose execution is invoked by means of a procedure call. (2) A set of related control statements that cause one or more programs to be performed.

**process.** A program or command that is actually running the computer. It consists of a loaded version of the executable file, its data, its stack, and its kernel data structures that represent the process's state within a multitasking environment. The executable file contains the machine instructions (and any calls to shared objects) that will be executed by the hardware. A process can contain multiple threads of execution.

The process is created with a **fork()** system call and ends using an **exit()** system call. Between **fork** and **exit**, the process is known to the system by a unique process identifier (PID).

Each process has its own virtual memory space and cannot access another process's memory directly. Communication methods across processes include pipes, sockets, shared memory, and message passing.

**prof.** A utility that produces an execution profile of an application or program. It is useful to identify which routines use the most CPU time. See the man page for **prof**.

**profiling.** The act of determining how much CPU time is used by each function or subroutine in a program. The histogram or table produced is called the execution profile.

**Program Marker Array.** An X-Windows run time monitor tool provided with parallel operating environment, used to provide immediate visual feedback on a program's execution.

**pthread.** A thread that conforms to the POSIX Threads Programming Model.

## R

**reduced instruction-set computer.** A computer that uses a small, simplified set of frequently-used instructions for rapid execution.

**reduction operation.** An operation, usually mathematical, that reduces a collection of data by one or more dimensions. For example, the arithmetic SUM operation is a reduction operation that reduces an array to a scalar value. Other reduction operations include MAXVAL and MINVAL.

**Reliable Scalable Cluster Technology.** A set of software components that together provide a comprehensive clustering environment for AIX. RSCT is the infrastructure used by a variety of IBM products to provide clusters with improved system availability, scalability, and ease of use.

**remote host.** Any host on a network except the one where a particular operator is working.

**remote shell (rsh).** A command supplied with both AIX and the IBM Parallel System Support Programs for AIX that lets you issue commands on a remote host.

**RISC.** See *reduced instruction-set computer*.

**RSCT.** See Reliable Scalable Cluster Technology.

**RSCT peer domain.** See *peer domain*.

## S

**shell script.** A sequence of commands that are to be executed by a shell interpreter such as the Bourne shell (**sh**), the C shell (**csh**), or the Korn shell (**ksh**). Script commands are stored in a file in the same format as if they were typed at a terminal.

**segmentation fault.** A system-detected error, usually caused by referencing an non-valid memory address.

**server.** A functional unit that provides shared services to workstations over a network — a file server, a print server, or a mail server, for example.

**signal handling.** A type of communication that is used by message passing libraries. Signal handling involves using AIX signals as an asynchronous way to move data in and out of message buffers.

**Single program, multiple data (SPMD).** A parallel programming model in which different processors execute the same program on different sets of data.

**source code.** The input to a compiler or assembler, written in a source language. Contrast with *object code*.

**source line.** A line of source code.

**SP.** IBM RS/6000 SP; a scalable system arranged in various physical configurations, that provides a high-powered computing environment.

**SPMD.** Single program, multiple data.

**standard input (STDIN).** In the AIX operating system, the primary source of data entered into a command. Standard input comes from the keyboard unless redirection or piping is used, in which case standard input can be from a file or the output from another command.

**standard output (STDOUT).** In the AIX operating system, the primary destination of data produced by a command. Standard output goes to the display unless redirection or piping is used, in which case standard output can go to a file or to another command.

**STDIN.** Standard input.

**STDOUT.** Standard output.

**stencil.** A pattern of memory references used for averaging. A 4-point stencil in two dimensions for a given array cell,  $x(i,j)$ , uses the four adjacent cells,  $x(i-1,j)$ ,  $x(i+1,j)$ ,  $x(i,j-1)$ , and  $x(i,j+1)$ .

**subroutine.** (1) A sequence of instructions whose execution is invoked by a call. (2) A sequenced set of instructions or statements that can be used in one or more computer programs and at one or more points in a computer program. (3) A group of instructions that can be part of another routine or can be called by another program or routine.

**synchronization.** The action of forcing certain points in the execution sequences of two or more asynchronous procedures to coincide in time.

**system administrator.** (1) The person at a computer installation who designs, controls, and manages the use of the computer system. (2) The person who is responsible for setting up, modifying, and maintaining the Parallel Environment.

**System Data Repository.** A component of the IBM Parallel System Support Programs for AIX software that provides configuration management for the SP system. It manages the storage and retrieval of system data across the control workstation, file servers, and nodes.

## T

**target application.** See *DPCL target application*.

**task.** A unit of computation analogous to an AIX process.

**thread.** A single, separately dispatchable, unit of execution. There can be one or more threads in a process, and each thread is executed by the operating system concurrently.

**tracing.** In PE, the collection of information about the execution of the program. This information is accumulated into a trace file that can later be examined.

**tracepoint.** Tracepoints are places in the program that, when reached during execution, cause the debugger to print information about the state of the program.

**trace record.** In PE, a collection of information about a specific event that occurred during the execution of your program. For example, a trace record is created for each send and receive operation that occurs in your program (this is optional and might not be appropriate). These records are then accumulated into a trace file that can later be examined.

## U

**unrolling loops.** See *loop unrolling*.

**user.** (1) A person who requires the services of a computing system. (2) Any person or any thing that can issue or receive commands and message to or from the information processing system.

| **User Space.** A version of the message passing library  
| that is optimized for direct access to the high  
| performance switch, that maximizes the performance  
| capabilities of the SP hardware.

**utility program.** A computer program in general support of computer processes; for example, a diagnostic program, a trace program, a sort program.

**utility routine.** A routine in general support of the processes of a computer; for example, an input routine.

## V

**variable.** (1) In programming languages, a named object that may take different values, one at a time. The values of a variable are usually restricted to one data type. (2) A quantity that can assume any of a given set of values. (3) A name used to represent a data item whose value can be changed while the program is running. (4) A name used to represent data whose value can be changed, while the program is running, by referring to the name of the variable.

**view.** (1) To display and look at data on screen. (2) A special display of data, created as needed. A view temporarily ties two or more files together so that the combined files can be displayed, printed, or queried. The user specifies the fields to be included. The original files are not permanently linked or altered; however, if the system allows editing, the data in the original files will be changed.

## **X**

**X Window System.** The UNIX industry's graphics windowing standard that provides simultaneous views of several executing programs or processes on high resolution graphics displays.

---

## Index

### Special characters

- bmaxdata 34
- buffer\_mem command-line flag 77, 81, 218
- clock\_source command-line flag 77
- css\_interrupt command-line flag 77
- eager\_limit command-line flag 217
- hints\_filtered command-line flag 78
- hostfile command-line flag 17
- infolevel command-line flag 39
- io\_buffer\_size command-line flag 81
- io\_errlog command-line flag 81
- ionodefile command-line flag 78
- msg\_api command-line flag 72
- polling\_interval command-line flag 78
- printenv command-line flag 84
- procs command-line flag 16
- q64 42
- qarch compiler option 33
- qextcheck compiler option 34
- qintsize 42
- retransmit\_interval command-line flag 79
- shared\_memory command-line flag 2, 15, 80
- single\_thread command-line flag 80
- stdoutmode command-line flag 31
- thread\_stacksize command-line flag 80
- udp\_packet\_size command-line flag 81
- wait\_mode command-line flag 81
- @bulkxfer 9

### Numerics

- 32-bit addressing 34
- 32-bit application 25, 33, 35, 37, 42
- 32-bit executable 15, 16
- 64-bit application 25, 33, 37, 42
- 64-bit executable 15

### A

- abbreviated names x
- accessibility 221
- acknowledgments 226
- acronyms for product names x
- address segments 34
- AIX function limitation 30
- AIX kernel thread 25
- AIX message catalog 33
- AIX profiling 11
- AIX signals 28
- AIXTHREAD\_SCOPE environment variable 38
- APIs
  - parallel task identification 145
- archive 13
- assorted constants 60
- asynchronous signals 28, 29
- atomic lock 41

### B

- bindings
  - subroutine 151, 175
- blocking collective 16, 30
- blocking communication 1
- blocking MPI call 41, 43
- blocking receive 4, 6
- blocking send 3, 6, 8
- buffering messages 217
- bulk transfer mode 9

### C

- C bindings 11
- C language binding datatypes 49
- C reduction function datatypes 50
- checkpoint
  - hang 41
- CHECKPOINT environment variable 39, 41
- checkpoint restrictions 39
- child process 41
- choice arguments 34
- clock source 34
- collective communication 15
- collective communication call 38, 41
- collective constants 60
- collective operations 62
- combiner constants 60
- command-line flags
  - buffer\_mem 77, 218
  - clock\_source 77
  - css\_interrupt 77
  - eager\_limit 78, 217
  - hints\_filtered 78
  - hostfile 17
  - infolevel 39
  - io\_buffer\_size 81
  - io\_errlog 81
  - ionodefile 78
  - msg\_api 72
  - polling\_interval 78
  - printenv 84
  - procs 16
  - retransmit\_interval 79
  - rexmit\_buf\_cnt 81
  - rexmit\_buf\_size 81
  - shared\_memory 2, 15, 80
  - single\_thread 80
  - stdoutmode 31
  - thread\_stacksize 80
  - udp\_packet\_size 81
  - wait\_mode 81
- command-line flags, POE 69
- commands
  - poe 25, 32
- communication stack 45
- communications adapter 9
- communicator 66
- constants
  - assorted 60

- constants (*continued*)
  - collective 60
  - collective operations 62
  - communicator and group comparisons 59
  - datatype decoding 60
  - derived datatypes 61
  - elementary datatype 61
  - empty group 62
  - environment inquiry keys 58
  - error classes 57
  - error handling specifiers 60
  - file operation 59
  - FORTRAN 90 datatype matching 63
  - maximum sizes 58
  - MPI 57
  - MPI-IO 59
  - null handles 62
  - one-sided 60
  - optional datatypes 62
  - predefined attribute keys 59
  - reduction function 61
  - special datatypes 61
  - threads 63
  - topologies 59
- construction of derived datatypes 61
- contention for processors 6
- conventions x
- core file generation 69
- credit flow control 5, 217, 218

## D

- datatype constructors 24
- datatype decoding functions 60
- datatypes
  - C language binding 49
  - C reduction functions 50
  - FORTRAN language bindings 50
  - FORTRAN reduction functions 51
  - predefined MPI 49
  - special purpose 49
- datatypes for reduction functions (C and C++) 61
- datatypes for reduction functions (FORTRAN) 61
- deadlock 19
- debugger restrictions 39
- DEVELOP mode 30
- diagnostic information 69
- disability 221
- dropped packets 10
- dynamic probe class library 39

## E

- eager limit 3, 5, 8, 65, 66, 217
- eager protocol 3, 5
- eager send 3, 5, 8, 66, 217, 218
- early arrival buffer 4, 5, 65, 217, 218
- early arrival list 4, 6
- elementary datatypes (C and C++) 61
- elementary datatypes (FORTRAN) 61
- empty group 62
- environment inquiry keys 58
- environment variables 1
  - AIXTHREAD\_SCOPE 38
  - CHECKPOINT 39, 41
  - LAPI\_USE\_SHM 34

- environment variables (*continued*)
  - MP\_ 33
  - MP\_ACK\_INTERVAL 44
  - MP\_ACK\_THRESH 9, 45, 76
  - MP\_BUFFER\_MEM 4, 5, 65, 66, 77, 218
  - MP\_CC\_SCRATCH\_BUFFER 9
  - MP\_CLOCK\_SOURCE 34, 77
  - MP\_CSS\_INTERRUPT 6, 21, 43, 44, 77
  - MP\_EAGER\_LIMIT 3, 5, 65, 66, 78, 217
  - MP\_EUIDEVELOP 30, 149, 151
  - MP\_EUIDEVICE 8
  - MP\_EUILIB 2, 3
  - MP\_HINTS\_FILTERED 23, 78
  - MP\_HOLD\_STDIN 32
  - MP\_HOSTFILE 16
  - MP\_INFOLEVEL 33, 39
  - MP\_INSTANCES 8
  - MP\_INTRDELAY 44
  - MP\_IO\_BUFFER\_SIZE 24, 81
  - MP\_IO\_ERRLOG 23, 81
  - MP\_IONODEFILE 20, 78
  - MP\_LAPI\_INET\_ADDR 17
  - MP\_MSG\_API 43, 72
  - MP\_PIPE\_SIZE 44
  - MP\_POLLING\_INTERVAL 7, 43, 78
  - MP\_PRINTENV 84
  - MP\_PRIORITY 10
  - MP\_PROCS 16, 65
  - MP\_RETRANSMIT\_INTERVAL 10, 79
  - MP\_REXMIT\_BUF\_CNT 7, 81
  - MP\_REXMIT\_BUF\_SIZE 7, 81
  - MP\_SHARED\_MEMORY 1, 2, 15, 30, 34, 80, 149, 151
  - MP\_SINGLE\_THREAD 7, 20, 21, 36, 38, 80
  - MP\_SNDBUF 31
  - MP\_STATISTICS 9, 10, 218
  - MP\_STDOUTMODE 31
  - MP\_SYNC\_ON\_CONNECT 44
  - MP\_TASK\_AFFINITY 10
  - MP\_THREAD\_STACKSIZE 37, 80
  - MP\_TIMEOUT 81
  - MP\_UDP\_PACKET\_SIZE 2, 45, 81
  - MP\_USE\_BULK\_XFER 9, 45
  - MP\_UTE 85
  - MP\_WAIT\_MODE 6, 81
  - MPI\_WAIT\_MODE 43
  - not recognized by PE 33
  - OBJECT\_MODE 42
  - reserved 33
- environment variables, POE 69
- error classes 57
- error handler 47
- error handling specifiers 60
- Ethernet adapter 16
- exit status
  - abnormal 29
  - continuing job step 27
  - normal 29
  - parallel application 26
  - terminating job step 27
  - values 26, 29, 37
- export file 13
- extended heap
  - specifying 35

## F

- file descriptor numbers 29
- file handle 66
- file operation constants 59
- flags, command-line
  - buffer\_mem 77, 218
  - clock\_source 77
  - css\_interrupt 77
  - eager\_limit 78, 217
  - hints\_filtered 78
  - hostfile 17
  - infolevel 39
  - io\_buffer\_size 81
  - io\_errlog 81
  - ionodefile 78
  - msg\_api 72
  - polling\_interval 78
  - printenv 84
  - procs 16
  - retransmit\_interval 79
  - rexmit\_buf\_cnt 81
  - rexmit\_buf\_size 81
  - shared\_memory 2, 15, 80
  - single\_thread 80
  - stdoutmode 31
  - thread\_stacksize 80
  - udp\_packet\_size 81
  - wait\_mode 81
- FORTTRAN 77 175
- FORTTRAN 90 175
- FORTTRAN 90 datatype matching constants 63
- FORTTRAN bindings 11, 12
- FORTTRAN language binding datatypes 50
- FORTTRAN reduction function dattypes 51
- function overloading 34
- functions
  - MPI 155

## G

- General Parallel File System (GPFS) 20
- gprof 11

## H

- hidden threads 21
- High Performance FORTRAN (HPF) 175
- hint filtering 23

## I

- I/O agent 20
- I/O node file 20
- IBM General Parallel File System (GPFS) 20
- IBM POWER4 server 10
- IBM POWER5 server 10
- import file 13
- Info objects 23
- ipcrm 16

## J

- job control 27
- Job Specifications 69
- job step progression 27

- job step termination 27
  - default 27

## K

- key collision 16
- key, value pair 23
- ksh93 30

## L

- language bindings
  - MPI 33
- LAPI 1, 17, 43, 44
  - sliding window protocol 4
  - used with MPI 43
- LAPI data transfer function 3
- LAPI dispatcher 4, 6, 9, 10
- LAPI parallel program 45
- LAPI protocol 25
- LAPI send side copy 7
- LAPI user message 7
- LAPI\_INIT 45
- LAPI\_TERM 45
- LAPI\_USE\_SHM environment variable 34
- limits, system
  - on size of MPI elements 65
- llcancel 16
- LoadLeveler 9, 26, 67
- LookAt message retrieval tool xi

## M

- M:N threads 38
- malloc and free 218
- MALLOCDEBUG 35
- MALLOCTYPE 35
- maximum sizes 58
- maximum tasks per node 67
- message address range 16
- message buffer 16, 25
- message credit 5, 65, 217, 218
- message descriptor 4
- message envelope 6
- message envelope buffer 65
- message packet transfer 7
- message passing
  - profiling 11
- message queue 42
- message retrieval tool, LookAt xi
- message traffic 9
- message transport mechanisms 1
- messages
  - buffering 217
- miscellaneous environment variables and flags 69
- mixed parallelism with MPI and threads 43
- MP\_ACK\_INTERVAL environment variable 44
- MP\_ACK\_THRESH environment variable 9, 45, 76
- MP\_BUFFER\_MEM environment variable 4, 5, 65, 66, 77, 81, 218
- MP\_CC\_SCRATCH\_BUFFER environment variable 9
- MP\_CLOCK\_SOURCE environment variable 34, 77
- MP\_CSS\_INTERRUPT environment variable 6, 21, 43, 44, 77
- MP\_EAGER\_LIMIT environment variable 3, 5, 65, 66, 217
- MP\_EUIDEVELOP environment variable 30, 149, 151
- MP\_EUIDEVICE environment variable 8

- MP\_EUILIB environment variable 2, 3
- MP\_HINTS\_FILTERED environment variable 23, 78
- MP\_HOLD\_STDIN environment variable 32
- MP\_HOSTFILE environment variable 16
- MP\_INFOLEVEL environment variable 33, 39
- MP\_INSTANCES environment variable 8
- mp\_intrdelay 44
- MP\_INTRDELAY environment variable 44
- MP\_IO\_BUFFER\_SIZE environment variable 24, 81
- MP\_IO\_ERRLOG environment variable 23, 81
- MP\_IONODEFILE environment variable 20, 78
- MP\_LAPI\_INET\_ADDR environment variable 17
- MP\_MSG\_API environment variable 43, 72
- MP\_PIPE\_SIZE environment variable 44
- MP\_POLLING\_INTERVAL environment variable 7, 43, 78
- MP\_PRINTENV environment variable 84
- MP\_PRIORITY environment variable 10
- MP\_PROCS environment variable 16, 65
- MP\_RETRANSMIT\_INTERVAL environment variable 10, 79
- MP\_REXMIT\_BUF\_CNT environment variable 7
- MP\_REXMIT\_BUF\_SIZE environment variable 7
- MP\_SHARED\_MEMORY environment variable 1, 2, 15, 30, 34, 80, 149, 151
- MP\_SINGLE\_THREAD environment variable 7, 20, 21, 36, 38, 80
- MP\_SNDBUF environment variable 31
- MP\_STATISTICS environment variable 9, 10, 218
- MP\_STDOUTMODE environment variable 31
- MP\_SYNC\_ON\_CONNECT environment variable 44
- MP\_TASK\_AFFINITY environment variable 10
- MP\_THREAD\_STACKSIZE environment variable 37, 80
- MP\_TIMEOUT environment variable 81
- MP\_UDP\_PACKET\_SIZE environment variable 2, 45, 81
- MP\_USE\_BULK\_XFER environment variable 9, 45
- MP\_UTE environment variable 85
- MP\_WAIT\_MODE environment variable 6, 81
- MPCI 44
- MPE subroutine bindings 151
- MPE subroutines 149
- MPI 69
  - functions 155
  - subroutines 155
  - used with LAPI 43
- MPI application exit without setting exit value 27
- MPI applications
  - performance 1
- MPI constants 57, 58, 59, 60, 61, 62, 63
- MPI datatype 19, 49
- MPI eager limit 66
- MPI envelope 7
- MPI internal locking 7
- MPI IP performance 2
- MPI library 37
  - architecture considerations 33
- MPI Library
  - performance 1
- MPI message size 7
- MPI reduction operations 53
- MPI size limits 65
- MPI subroutine bindings 175
- MPI wait call 1, 3, 4, 6
- MPI\_Abort 26, 28
- MPI\_ABORT 26, 28
- MPI\_File 19
- MPI\_File object 22
- MPI\_Finalize 27
- MPI\_FINALIZE 27, 38, 45

- MPI\_INIT 38, 45
- MPI\_INIT\_THREAD 38
- MPI\_THREAD\_FUNNELED 38
- MPI\_THREAD\_MULTIPLE 38
- MPI\_THREAD\_SINGLE 38
- MPI\_WAIT\_MODE environment variable 43
- MPI\_WTIME\_IS\_GLOBAL 34
- MPI-IO
  - API user tasks 20
  - considerations 20
  - data buffer size 24
  - datatype constructors 24
  - deadlock prevention 19
  - definition 19
  - error handling 22
  - features 19
  - file interoperability 24
  - file management 21
  - file open 21
  - file tasks 21
  - hidden threads 21
  - I/O agent 20
  - Info objects 23
  - logging errors 23
  - portability 19
  - robustness 19
  - versatility 19
- MPI-IO constants 59
- MPL 25
  - not supported 25
- mpxlf\_r 13
- multi-chip module (MCM) 10
- mutex lock 43

## N

- n-task parallel job 25
- named pipes 32
- non-blocking collective 16, 30
- non-blocking collective communication subroutines 149
- non-blocking communication 1
- non-blocking receive 4
- non-blocking send 3
- null handles 62

## O

- OBJECT\_MODE environment variable 42
- OK to send response 6
- one-sided constants 60
- one-sided message passing API 1, 7, 23, 36, 38, 43, 45, 46, 155, 205
- op operation
  - datatypes 54
- operations
  - predefined 53
  - reduction 53
- optional datatypes 62

## P

- packet sliding window 9
- packet statistics 10
- parallel application 45
- parallel I/O 19
- parallel job 25

- parallel job termination 29
- parallel task I/O 69
- parallel task identification API
  - subroutines 145
- parallel utility subroutines 87
  - MP\_BANDWIDTH 91
  - MP\_DISABLEINTR 96
  - MP\_ENABLEINTR 99
  - MP\_FLUSH 102
  - MP\_INIT\_CKPT 104
  - MP\_QUERYINTR 106
  - MP\_QUERYINTRDELAY 109
  - MP\_SET\_CKPT\_CALLBACKS 110
  - MP\_SETINTRDELAY 113
  - MP\_STATISTICS\_WRITE 114
  - MP\_STATISTICS\_ZERO 117
  - MP\_STDOUT\_MODE 118
  - MP\_STDOUTMODE\_QUERY 121
  - MP\_UNSET\_CKPT\_CALLBACKS 123
- mpc\_bandwidth 91
- mpc\_disableintr 96
- mpc\_enableintr 99
- mpc\_flush 102
- mpc\_init\_ckpt 104
- mpc\_isatty 89
- mpc\_queryintr 106
- mpc\_queryintrdelay 109
- mpc\_set\_ckpt\_callbacks 110
- mpc\_setintrdelay 113
- mpc\_statistics\_write 114
- mpc\_statistics\_zero 117
- mpc\_stdout\_mode 118
- mpc\_stdoutmode\_query 121
- mpc\_unset\_ckpt\_callbacks 123
- pe\_dbg\_breakpoint 125
- pe\_dbg\_checkpoint 131
- pe\_dbg\_checkpoint\_wait 135
- pe\_dbg\_getcrd 137
- pe\_dbg\_getrtid 138
- pe\_dbg\_getvtid 139
- pe\_dbg\_read\_cr\_errfile 140
- pe\_dbg\_restart 141
- Partition Manager 69
- Partition Manager Daemon (PMD) 16, 25, 29, 30, 31, 32
- PCI adapter 67
- PE 3.2 44
- PE 4.1 44
- PE co-scheduler 10
- performance
  - shared memory 16
- pipes 39, 41
  - STDIN, STDOUT, or STDERR 32
  - STDIN, STDOUT, or STDERR, output 32
- pmd 32
- POE
  - argument limits 31
  - program argument 31
  - shell script 30
  - user applications 25
- poe command 25, 32
- POE command-line flags 69
  - ack\_thresh 76
  - adapter\_use 70
  - buffer\_mem 77, 218
  - bulk\_min\_msg\_size 80
  - clock\_source 77
  - cmdfile 73

- POE command-line flags *(continued)*
  - coredir 83
  - corefile\_format 83
  - corefile\_format\_sigterm 83
  - cpu\_use 70
  - css\_interrupt 77
  - debug\_notimeout 76
  - eager\_limit 78, 217
  - euidevelop 69, 84
  - euidevice 70
  - euilib 70
  - euilibpath 70
  - hfile 71
  - hints\_filtered 78
  - hostfile 17, 71
  - ilevel 69, 76
  - infolevel 39, 69, 76
  - instances 71
  - io\_buffer\_size 81
  - io\_errlog 81
  - ionodefile 78
  - labelio 74
  - llfile 73
  - msg\_api 72
  - newjob 73
  - nodes 72
  - pgmmode 73
  - pmdlog 76
  - polling\_interval 78
  - printenv 84
  - procs 16, 71
  - pulse 71
  - resd 71
  - retransmit\_interval 79
  - retry 71
  - retrycount 71
  - rexmit\_buf\_cnt 81
  - rexmit\_buf\_size 81
  - rmpool 72
  - save\_llfile 73
  - savehostfile 72
  - shared\_memory 2, 15, 80
  - single\_thread 80
  - stdinmode 75
  - stdoutmode 31, 75
  - task\_affinity 74
  - tasks\_per\_node 72
  - thread\_stacksize 80
  - udp\_packet\_size 81
  - use\_bulk\_xfer 79
  - wait\_mode 81
- POE considerations
  - 64-bit application 42
  - AIX 37
  - AIX function limitation 30
  - AIX message catalog considerations 33
  - architecture 33
  - automount daemon 30
  - checkpoint and restart 39
  - child task 37
  - collective communication call 38
  - entry point 37
  - environment overview 25
  - exit status 26
  - exits, abnormal 29
  - exits, normal 29
  - exits, parallel task 29

POE considerations (*continued*)

- file descriptor numbers 29
- fork limitations 37
- job step default termination 27
- job step function 27
- job step progression 27
- job step termination 27
- job termination 29
- language bindings 33
- large numbers of tasks 35
- LoadLeveler 26
- M:N threads 38
- MALLOCDEBUG 35
- mixing collective 30
- MPI\_INIT 38
- MPI\_INIT\_THREAD 38
- MPI\_WAIT\_MODE 43
- network tuning 31
- nopoll 43
- order requirement for system includes 38
- other differences 45
- parent task 37
- POE additions 27
- remote file system 30
- reserved environment variables 33
- root limitation 30
- shell scripts 30
- signal handler 28
- signal library 25
- single threaded 36
- STDIN, STDOUT, or STDERR 30, 32
- STDIN, STDOUT, or STDERR, output 31
- STDIN, STDOUT, or STDERR, rewinding 30
- task initialization 37
- thread stack size 37
- thread termination 37
- thread-safe libraries 37
- threads 36
- user limits 26
- user program, passing string arguments 31
- using MPI and LAPI together 43
- virtual memory segments 34

POE environment variables 69

- MP\_ACK\_INTERVAL 44
- MP\_ACK\_THRESH 9, 45, 76
- MP\_ADAPTER\_USE 70
- MP\_BUFFER\_MEM 4, 5, 65, 66, 77, 218
- MP\_BULK\_MIN\_MSG\_SIZE 80
- MP\_CC\_SCRATCH\_BUFFER 9
- MP\_CKPTDIR 73
- MP\_CKPTFILE 73
- MP\_CLOCK\_SOURCE 34, 77
- MP\_CMDFILE 73
- MP\_COREDIR 83
- MP\_COREFILE\_FORMAT 83
- MP\_COREFILE\_SIGTERM 83
- MP\_CPU\_USE 70
- MP\_CSS\_INTERRUPT 6, 21, 43, 44, 77
- MP\_DBXPROMPTMOD 83
- MP\_DEBUG\_INITIAL\_STOP 76
- MP\_DEBUG\_NOTIMEOUT 76
- MP\_EAGER\_LIMIT 3, 5, 65, 66, 78, 217
- MP\_EUIDEVELOP 30, 84, 149, 151
- MP\_EUIDEVICE 8, 70
- MP\_EUILIB 2, 3, 70
- MP\_EUILIBPATH 70
- MP\_FENCE 84

POE environment variables (*continued*)

- MP\_HINTS\_FILTERED 23, 78
- MP\_HOLD\_STDIN 32, 75
- MP\_HOSTFILE 16, 71
- MP\_INFOLEVEL 33, 39, 76
- MP\_INSTANCES 8, 71
- MP\_INTRDELAY 44
- MP\_IO\_BUFFER\_SIZE 24, 81
- MP\_IO\_ERRLOG 23, 81
- MP\_IONODEFILE 20, 78
- MP\_LABELIO 74
- MP\_LAPI\_INET\_ADDR 17
- MP\_LLFILE 73
- MP\_MSG\_API 43, 72
- MP\_NEWJOB 73
- MP\_NOARGLIST 84
- MP\_NODES 72
- MP\_PGMMODEL 73
- MP\_PIPE\_SIZE 44
- MP\_PMDLOG 76
- MP\_POLLING\_INTERVAL 7, 43, 78
- MP\_PRINTENV 84
- MP\_PRIORITY 10, 84
- MP\_PROCS 16, 65, 71
- MP\_PULSE 71
- MP\_REMOTEDIR 72
- MP\_RESD 71
- MP\_RETRANSMIT\_INTERVAL 10, 79
- MP\_RETRY 71
- MP\_RETRYCOUNT 71
- MP\_REXMIT\_BUF\_CNT 7, 81
- MP\_REXMIT\_BUF\_SIZE 7, 81
- MP\_RMPOOL 72
- MP\_SAVE\_LLFILE 73
- MP\_SAVEHOSTFILE 72
- MP\_SHARED\_MEMORY 1, 2, 15, 30, 34, 80, 149, 151
- MP\_SINGLE\_THREAD 7, 20, 21, 36, 38, 80
- MP\_SNDBUF 31
- MP\_STATISTICS 9, 10, 218
- MP\_STDINMODE 75
- MP\_STDOUTMODE 31, 75
- MP\_SYNC\_ON\_CONNECT 44
- MP\_TASK\_AFFINITY 10, 74
- MP\_TASKS\_PER\_NODE 72
- MP\_THREAD\_STACKSIZE 37, 80
- MP\_TIMEOUT 72, 81
- MP\_UDP\_PACKET\_SIZE 2, 45, 81
- MP\_USE\_BULK\_XFER 9, 45, 79
- MP\_UTE 85
- MP\_WAIT\_MODE 6, 81
- MPI\_WAIT\_MODE 43

POE threads 45

- point-to-point communications 3
- point-to-point messages 15
- polling considerations 6
- predefined attribute keys 59
- predefined error handler 47
- predefined MPI datatype 49
- process contention scope 38
- process profiling 41
- prof 11
- profiling
  - counts 11
  - export file 12
  - library 12
  - message passing 11
  - MPI nameshift 11

- profiling (*continued*)
  - shared library 12
- profiling library 11
- profiling MPI routines 12
- program exit without setting exit value 27
- programming considerations
  - user applications 25
- protocol striping 8
- pSeries high performance switch 67
- pthread lock 41

## Q

- quotation marks 31

## R

- receive buffer 9
- reduction operations
  - C example 55
  - datatype arguments 53
  - examples 55
  - FORTTRAN example 55
  - MPI 53
  - predefined 53
- Remote Direct Memory Access (RDMA) 9
- rendezvous message 6
- rendezvous protocol 3, 5, 66, 217, 218
- reserved environment variables 33
- resource limits 26
- restart restrictions 40
- results of communicator and group comparisons 59
- retransmission buffer 7, 8
- return code 19
- rewinding STDIN, STDOUT, or STDERR 30
- root limitation 30
- rtl\_enable 13

## S

- sa\_sigaction 28
- scratch buffer 9
- semaphore 42
- send buffer 9
- service thread 6, 45
- setuid program 39
- shared memory 1, 2, 15, 30, 39, 41, 42, 67, 217
  - reclaiming 16
- shared memory key collision 16
- shared memory performance considerations 16
- shmat 39
- shmget 34
- sigaction 28
- SIGALRM 29
- SIGIO 29
- signal handler 28, 36
  - POE 28
  - user defined 28, 29
- signal library 25
- SIGPIPE 29
- sigwait 28
- Simultaneous Multi-Threading (SMT) 67
- single thread considerations 6
- single threaded applications 36
- sockets 41
- special datatypes 61

- special purpose datatypes 49
- striping 8
- subroutine bindings 151, 175
  - collective communication 175
  - communicator 179
  - conversion functions 182
  - derived datatype 183
  - environment management 189
  - external interfaces 191
  - group management 193
  - Info object 195
  - memory allocation 196
  - MPI-IO 197, 204
  - non-blocking collective communication 151
  - one-sided communication 205
  - point-to-point communication 208
  - profiling control 213
  - topology 214
- subroutines
  - MPE 149
  - MPI 155
  - non-blocking collective communication 149
  - parallel task identification API 145
  - parallel utility subroutines 87
  - poe\_master\_tasks 146
  - poe\_task\_info 147
- switch clock 34
- system contention scope 38
- system limits
  - on size of MPI elements 65

## T

- tag 66
- task limits 67
- task synchronization 25
- thread context 6
- thread stack size
  - default 37
- thread-safe library 7
- threaded MPI library 25
- threaded programming 36
- threads and mixed parallelism with MPI 43
- threads constants 63
- threads library 25
- threads library considerations
  - AIX signals 28
- topologies 59
- trademarks 225
- tuning parameter
  - sb\_max 2
  - udp\_recvspace 2
  - udp\_sendspace 2

## U

- UDP ports 8
- UDP/IP 2, 4
- UDP/IP transport 2
- unacknowledged packets 10
- unsent data 66
- upcall 4, 6
- user resource limits 26
- User Space 1, 2, 4
- User Space FIFO mechanism 8
- User Space FIFO packet buffer 8

- User Space library 1
- User space protocol 43
- User Space transport 2, 3, 6
- User Space window 3

## **V**

- virtual address space 9
- virtual memory segments 34

## **W**

- wait
  - MPI 1, 2, 3, 4, 6
- window 66

## **X**

- xprofiler 11

---

## Readers' comments – We'd like to hear from you

IBM Parallel Environment for AIX 5L  
MPI Programming Guide  
Version 4 Release 2

Publication No. SA22-7945-02

Overall, how satisfied are you with the information in this book?

	Very Satisfied	Satisfied	Neutral	Dissatisfied	Very Dissatisfied
Overall satisfaction	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>

How satisfied are you that the information in this book is:

	Very Satisfied	Satisfied	Neutral	Dissatisfied	Very Dissatisfied
Accurate	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
Complete	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
Easy to find	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
Easy to understand	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
Well organized	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
Applicable to your tasks	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>

Please tell us how we can improve this book:

Thank you for your responses. May we contact you? ☐ Yes ☐ No

When you send comments to IBM, you grant IBM a nonexclusive right to use or distribute your comments in any way it believes appropriate without incurring any obligation to you.

---

Name

---

Address

---

Company or Organization

---

Phone No.



Cut or Fold  
Along Line

Fold and Tape

Please do not staple

Fold and Tape



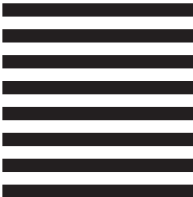
NO POSTAGE  
NECESSARY  
IF MAILED IN THE  
UNITED STATES

**BUSINESS REPLY MAIL**

FIRST-CLASS MAIL PERMIT NO. 40 ARMONK, NEW YORK

POSTAGE WILL BE PAID BY ADDRESSEE

IBM Corporation  
Department 55JA, Mail Station P384  
2455 South Road  
Poughkeepsie, NY  
12601-5400



Fold and Tape

Please do not staple

Fold and Tape

Cut or Fold  
Along Line





Program Number: 5765-F83

SA22-7945-02

